

1. Basics of Information

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS

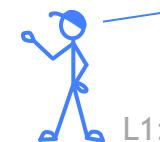
What is “Information”?

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”



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

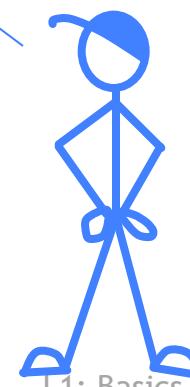
Information received when learning that choice was x_i :

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

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)



Information Conveyed by Data

Even when data doesn't resolve all the uncertainty

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

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

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

Example: Information Content

Examples:

- information in one coin flip:

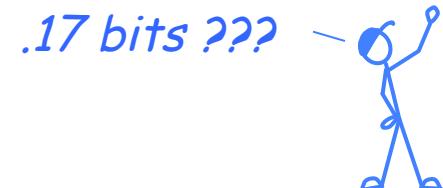
$$N = 2 \quad M = 1 \quad \text{Info content} = \log_2(2/1) = 1 \text{ bit}$$

- 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:

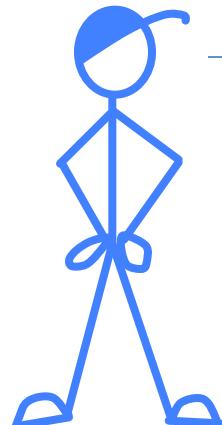
$$N = 36 \quad M = 1 \quad \text{Info content} = \log_2(36/1) = 5.17$$



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.*

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

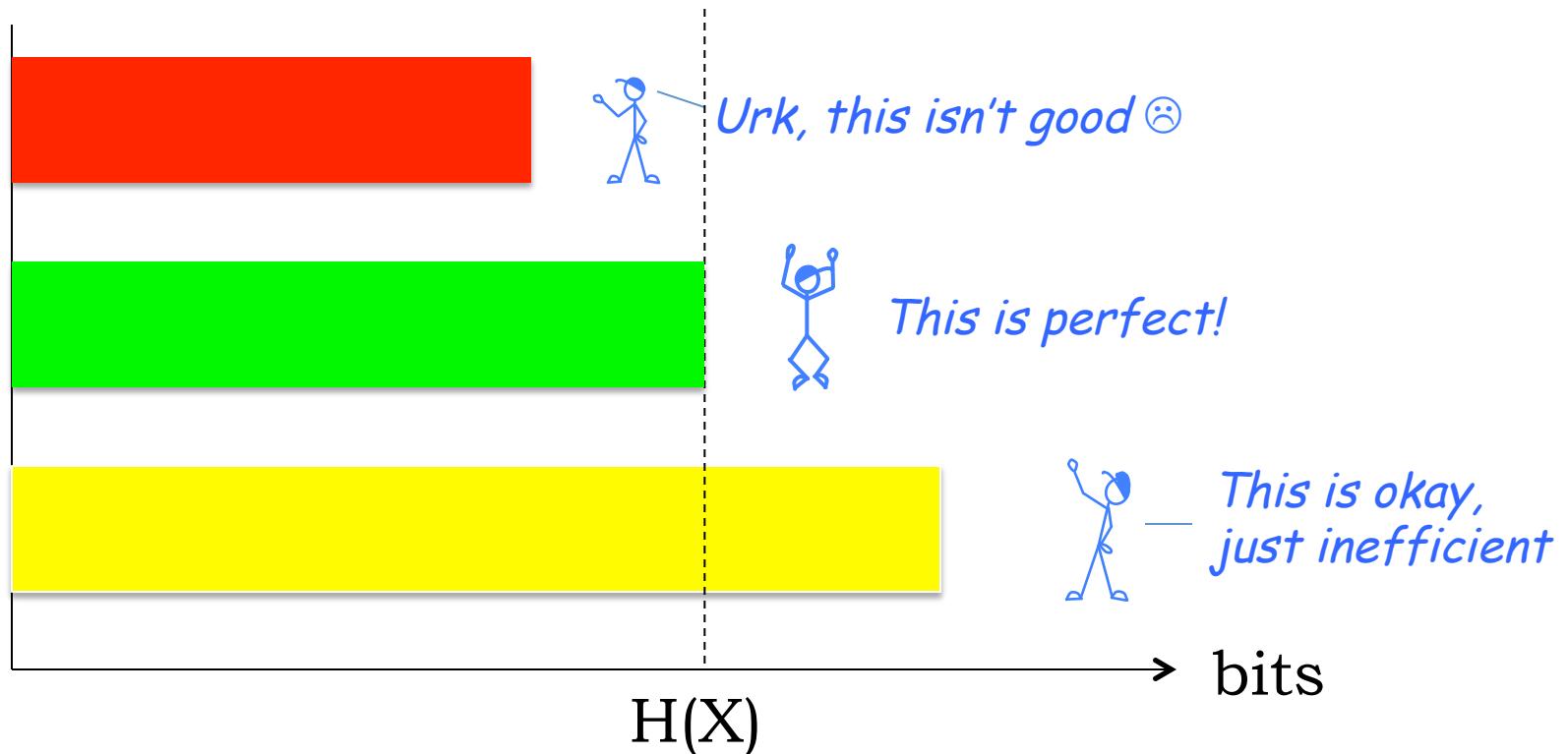
$choice_i$	p_i	$\log_2(1/p_i)$
“A”	1/3	1.58 bits
“B”	1/2	1 bit
“C”	1/12	3.58 bits
“D”	1/12	3.58 bits

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

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



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
0	1	10	11	0 1 1 0  ABBA? ABC? ADA?

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.

Encoding

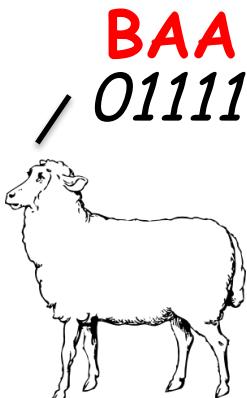
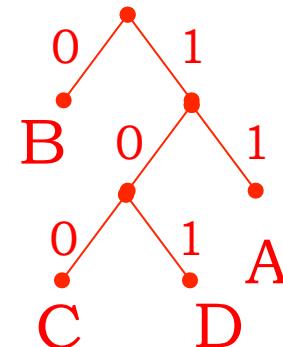
$B \leftrightarrow 0$

$A \leftrightarrow 11$

$C \leftrightarrow 100$

$D \leftrightarrow 101$

Binary tree

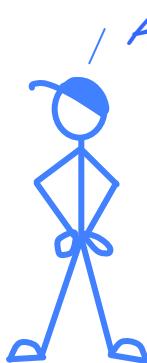
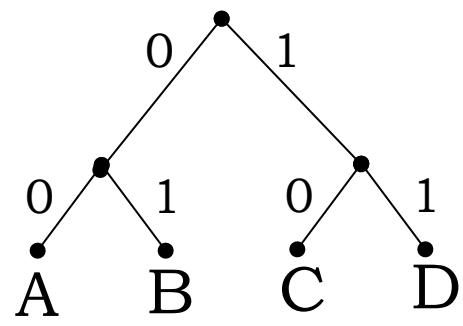


BAA

/ 01111

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

$$\sum_{i=1}^N \left(\frac{1}{N}\right) \log_2 \left(\frac{1}{\frac{1}{N}} \right) = \log_2(N)$$

Examples:

Fixed-length are often a little inefficient...



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

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

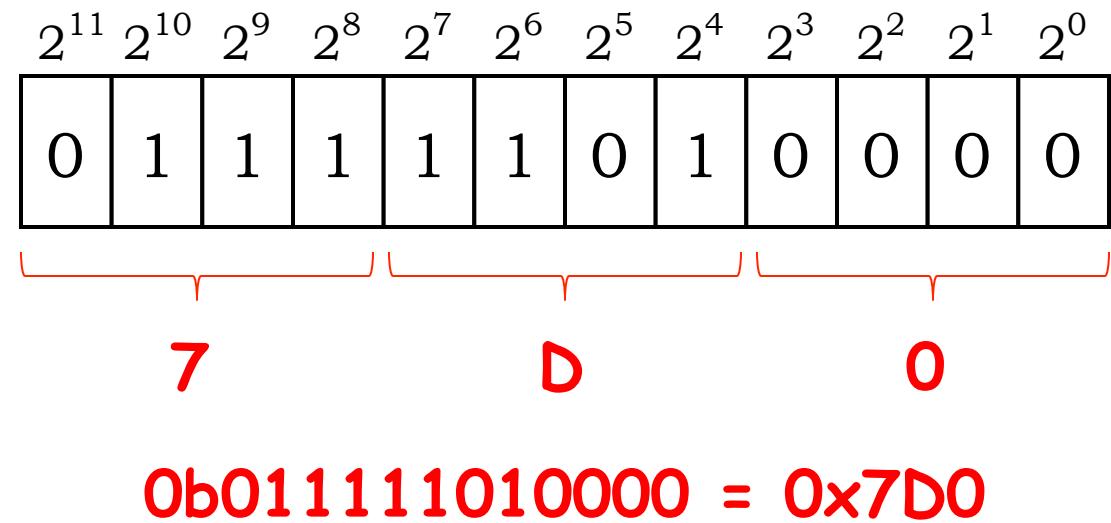
Largest number: $2^N - 1$

Hexadematical 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.

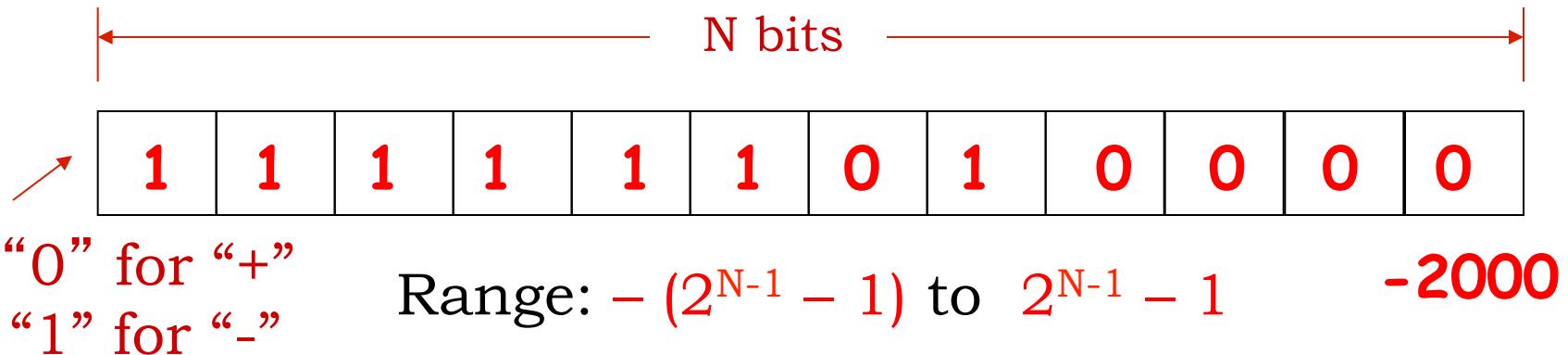
Hexadecimal - base 16	
0000	- 0
0001	- 1
0010	- 2
0011	- 3
0100	- 4
0101	- 5
0110	- 6
0111	- 7
1000	- 8
1001	- 9
1010	- A
1011	- B
1100	- C
1101	- D
1110	- E
1111	- F



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).

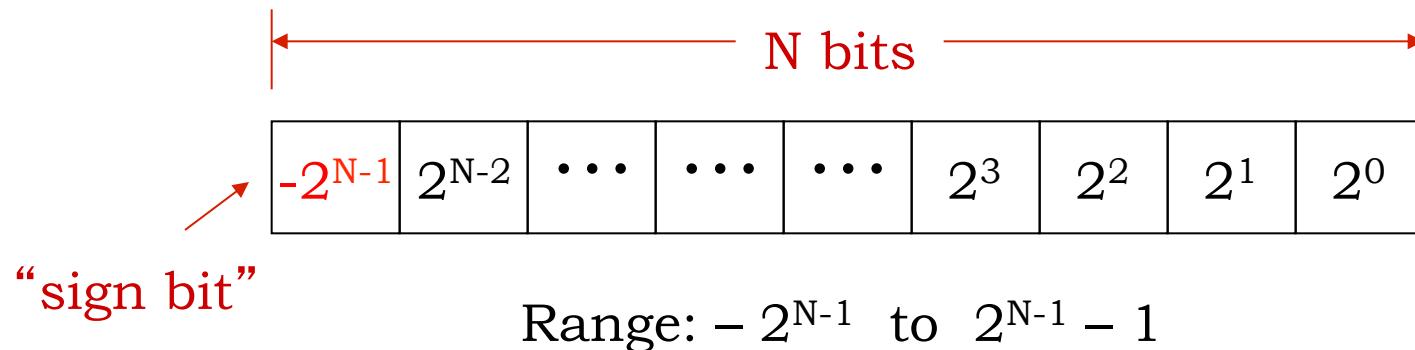
We could adopt that approach for binary representations:



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

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: $00\dots0000$ 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!

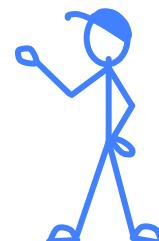
- To compute $B-A$, we'll just use addition and compute $B+(-A)$. But how do we figure out the representation for $-A$?

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

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

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

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



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:

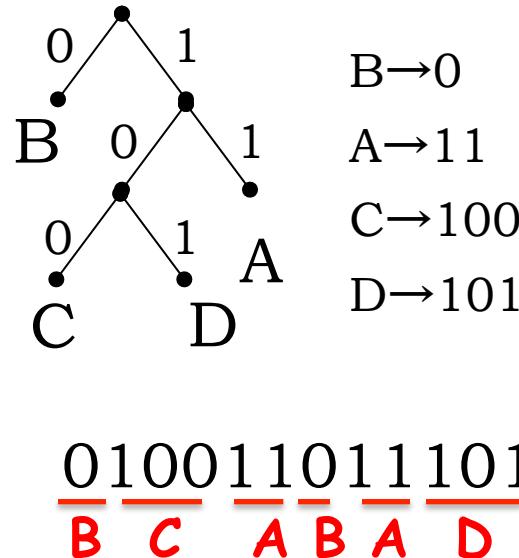
- Higher probability → shorter encodings
- Lower probability → longer encodings

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

Example

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

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



High probability,
Less information

Low probability,
More information

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

Huffman's Algorithm

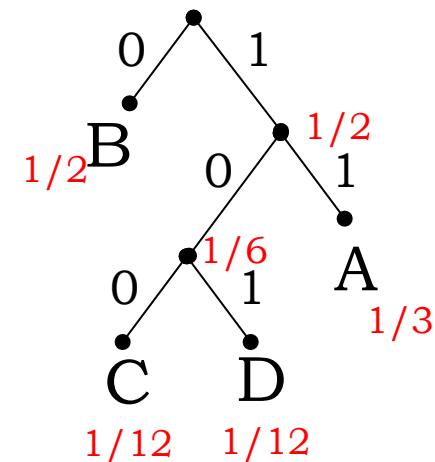
Given a set of symbols and their probabilities, constructs an optimal variable-length encoding.

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



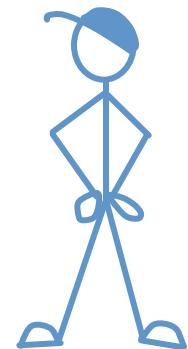
Can We Do Better?

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

Error Detection and Correction

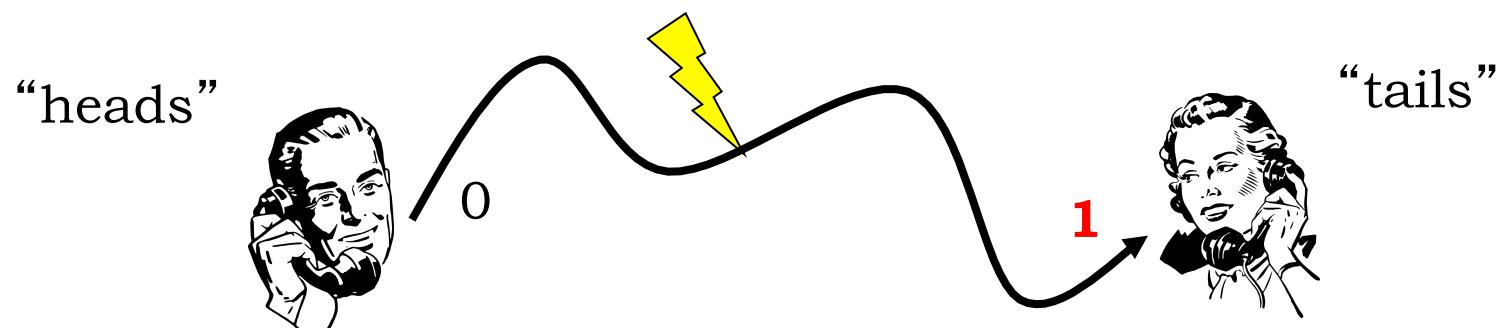
Suppose we wanted to reliably transmit the result of a single coin flip:

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”.



Hamming Distance

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

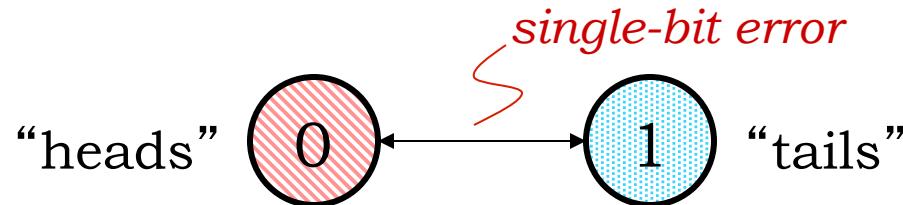
01	1	00	10
↑↑			
01	0	01	10



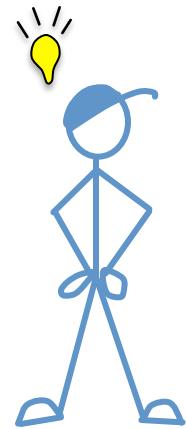
Hamming Distance & Bit Errors

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

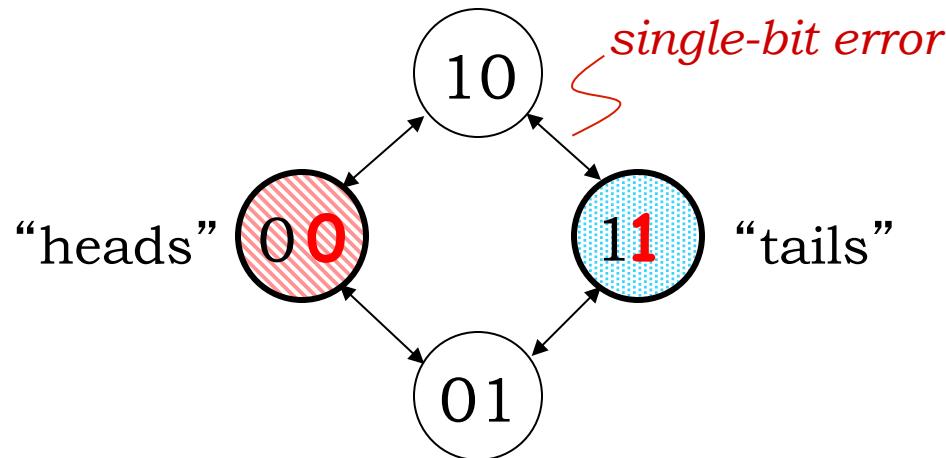
The problem with our simple encoding is that the two valid code words (“0” and “1”) also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word...



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 $\min \text{HD}(\text{code words} + \text{parity}) = 2$.

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

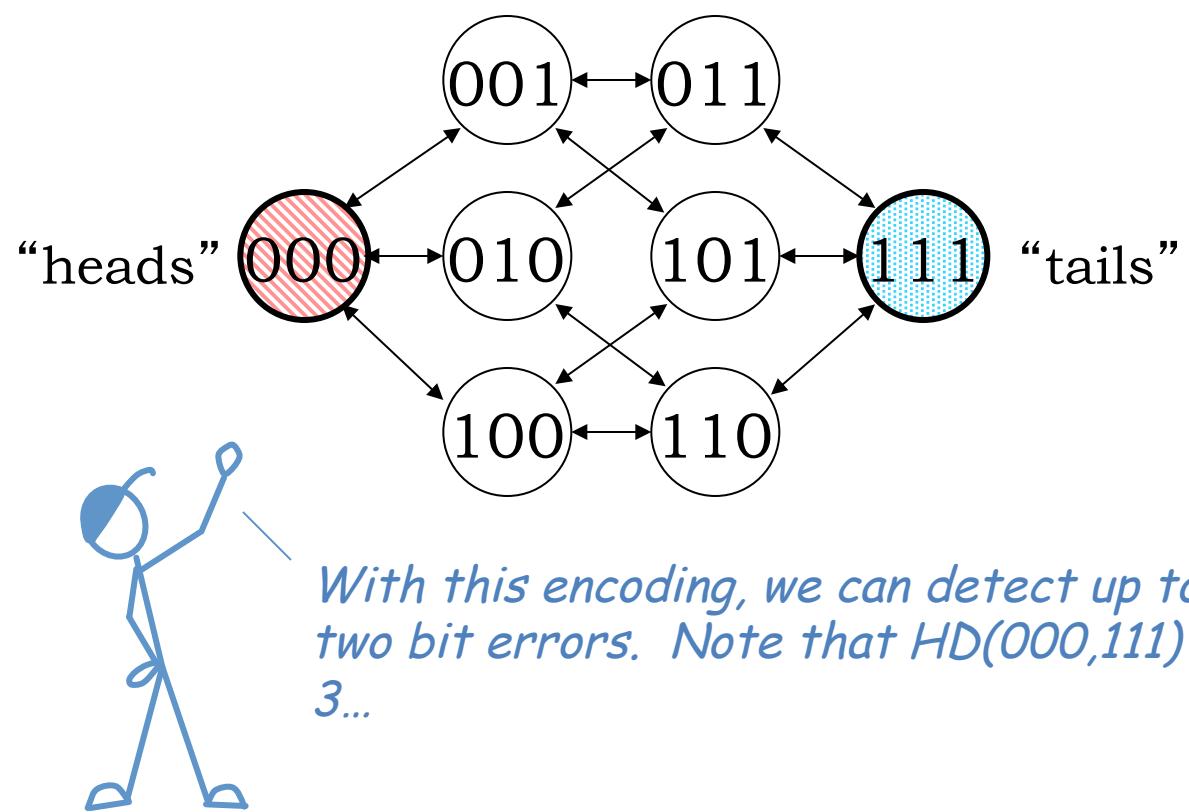
0 1 1 0 0 **0** 0 1 0 0 1 1 → single-bit error (detected)

0 1 1 0 0 **0** **1** 1 0 0 1 1 → 2-bit error (not detected)

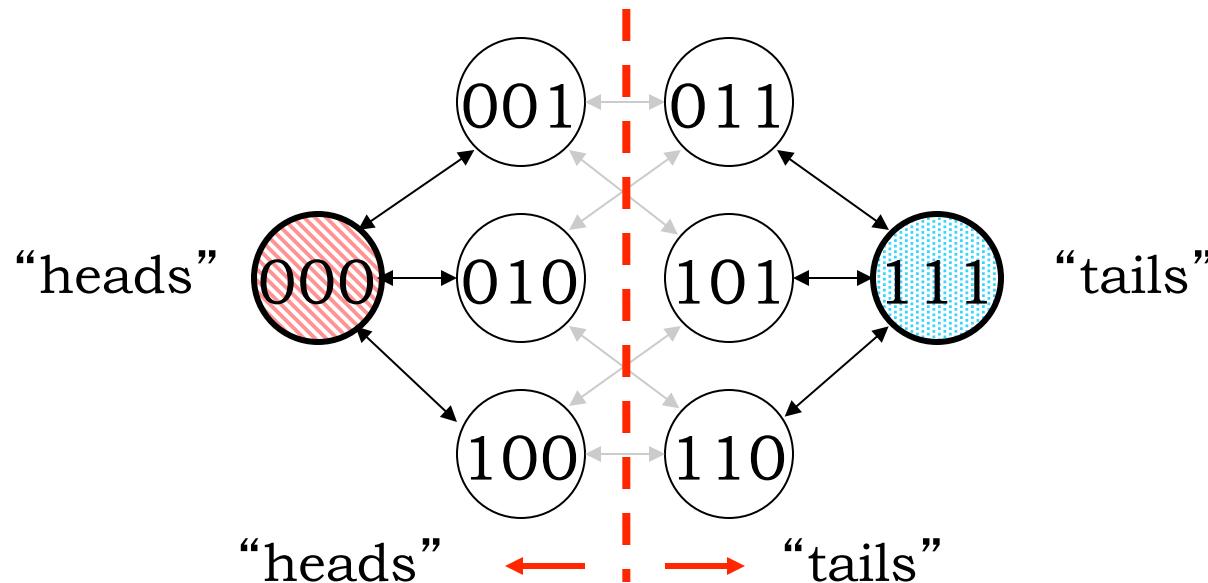
- One can “count” by summing the bits in the word modulo 2 (which is equivalent to XOR’ing the bits together).

Detecting Multi-bit Errors

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



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.

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

2. The Digital Abstraction

6.004.1x Computation Structures
Part 1 – Digital Circuits

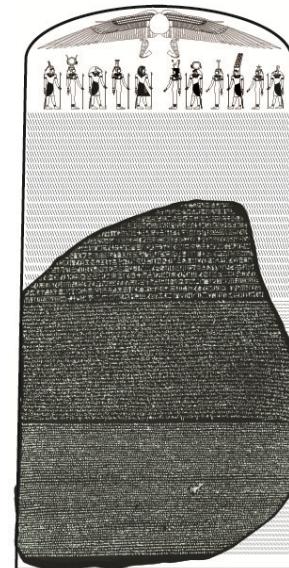
Copyright © 2015 MIT EECS

Concrete Encoding of Information

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

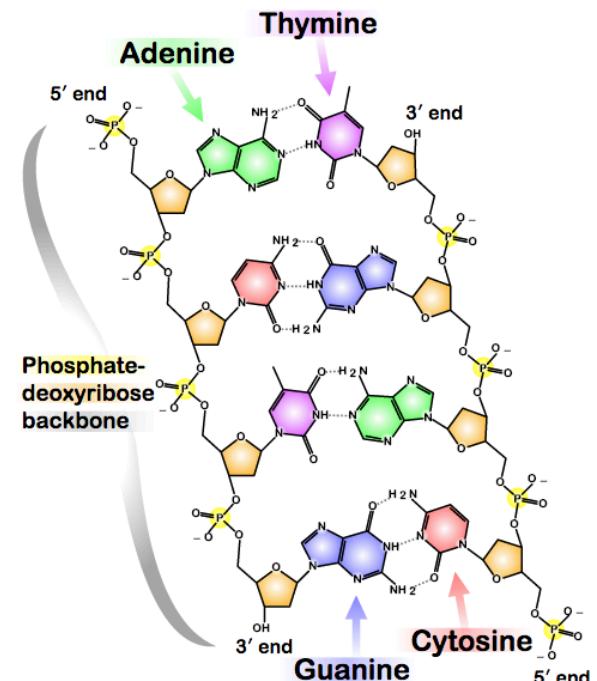
If we're going to design a machine that manipulates information, how should that information be physically encoded?

Rosetta Stone



Captmondo (CC BY-SA 3.0)

DNA



Madeleine Price Ball (CC BY-SA 3.0)

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)

Let's Use Electrical Phenomenon...

Consider using phenomenon associated with charged particles:

voltages	phase
currents	frequency

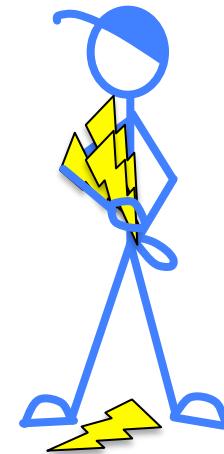
In this course we'll use **voltages** to encode information. But the best choice depends on the intended application...

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



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?

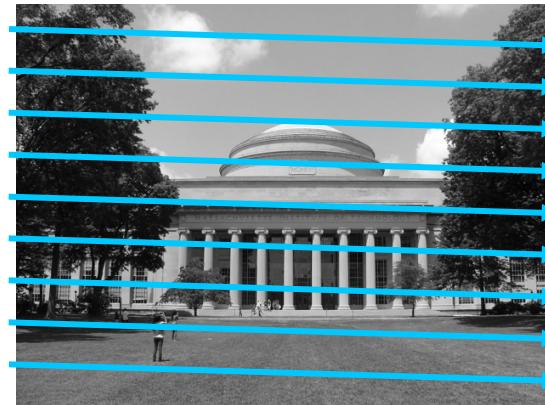
Suppose we can reliably distinguish voltages that differ by $1/2^N$ volts. Then we can represent N bits of information by voltages in the range 0V to 1V. What are realistic values for N?

Using Voltages to Encode a Picture

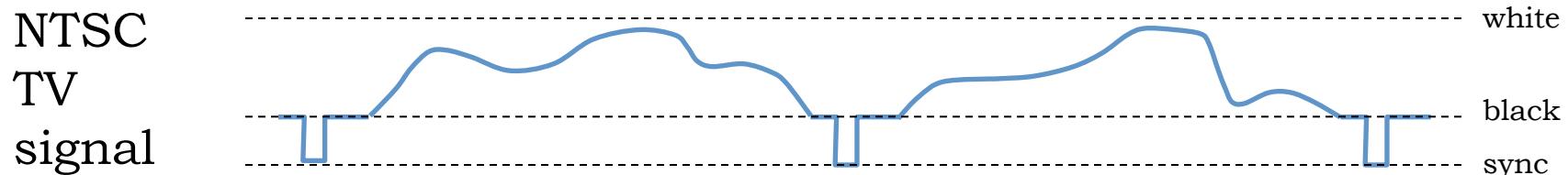
Representation of a picture:

Scan points in some prescribed raster order...

Generate voltage waveform:



John Phelan (CC BY 3.0)

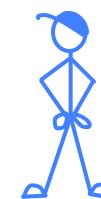


Information Processing = Computation



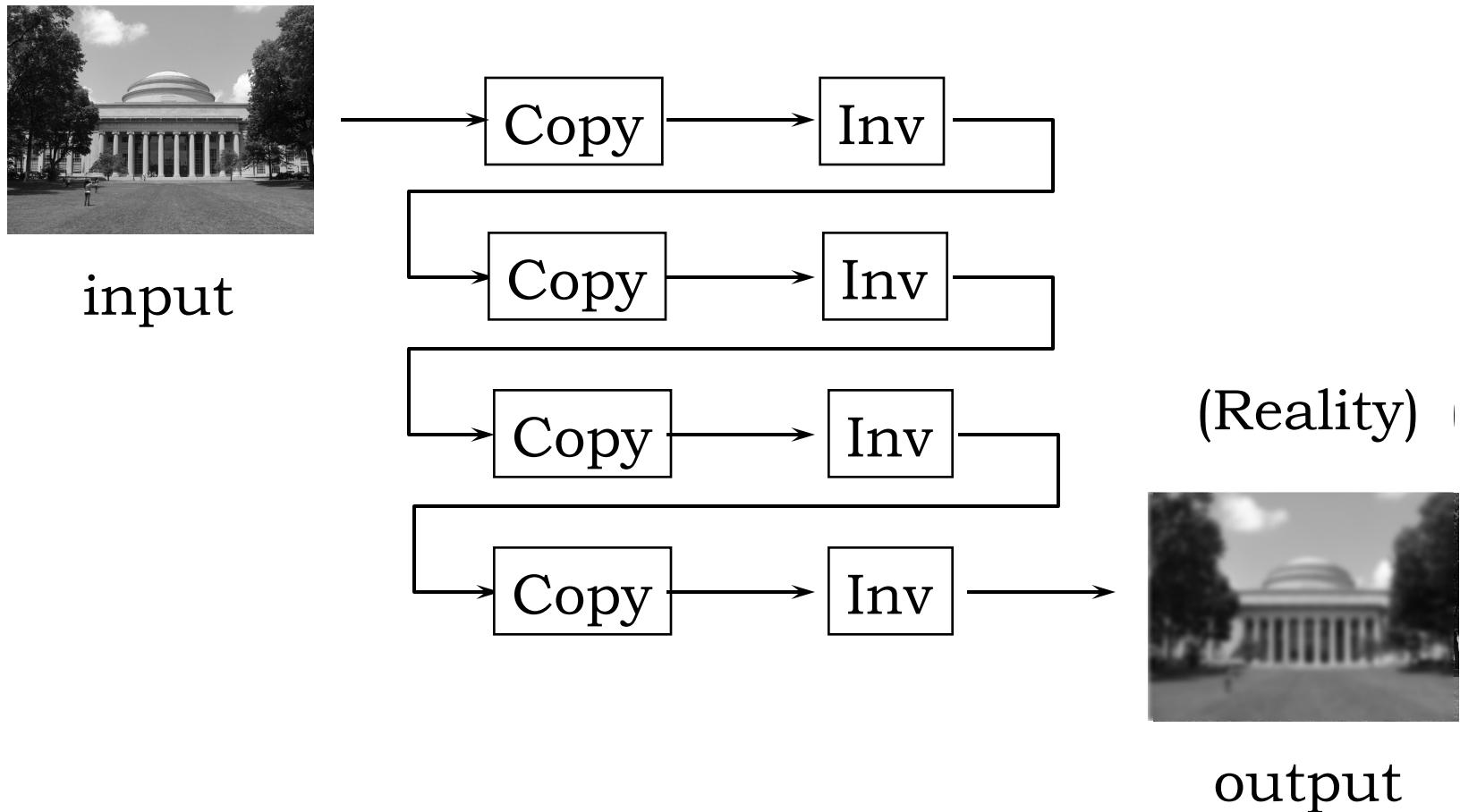
Why have processing blocks?

- 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!

Let's Build a System!



Why Did Our System Fail?

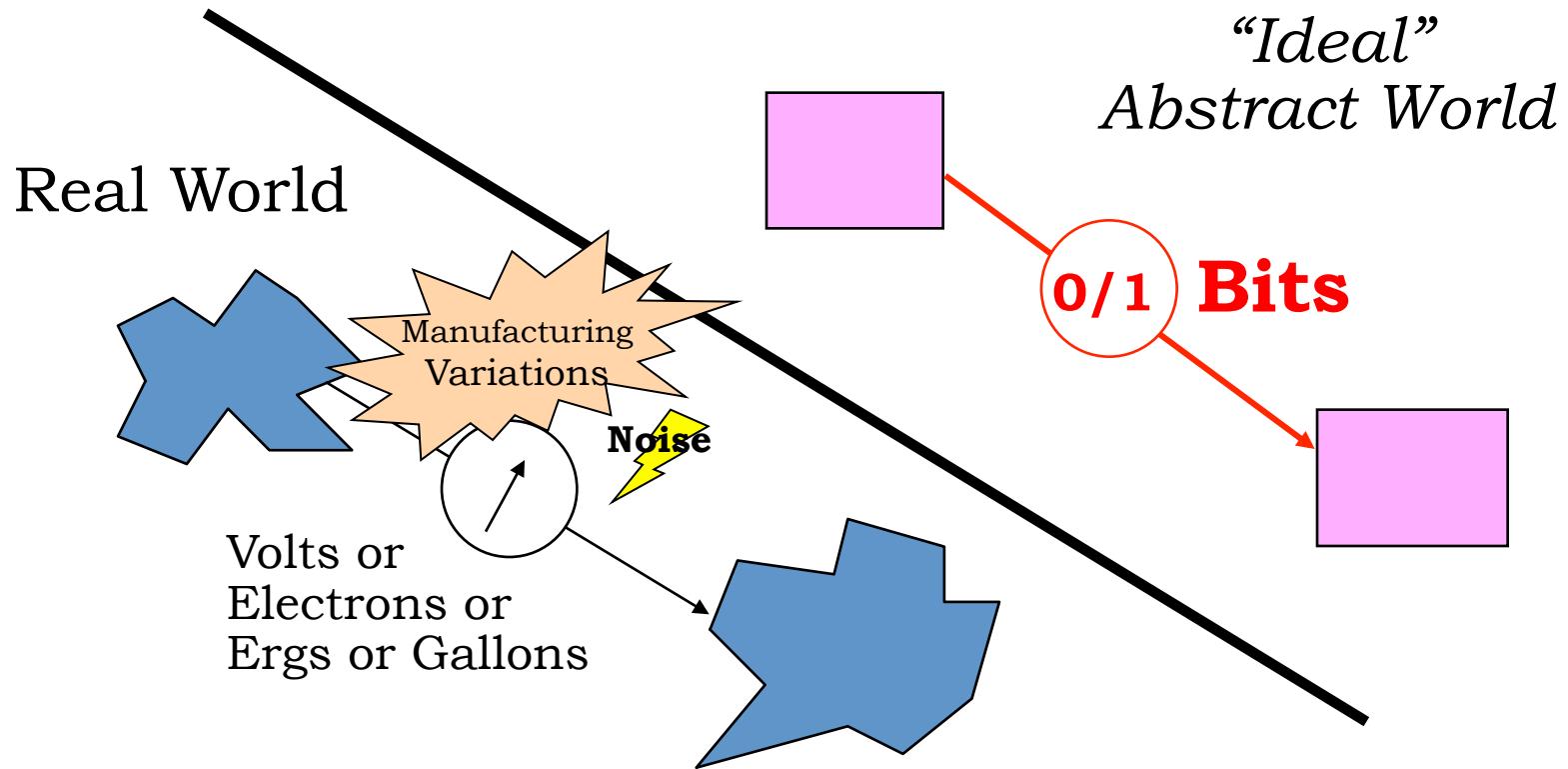
Why doesn't theory match reality?

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.

The Digital Abstraction



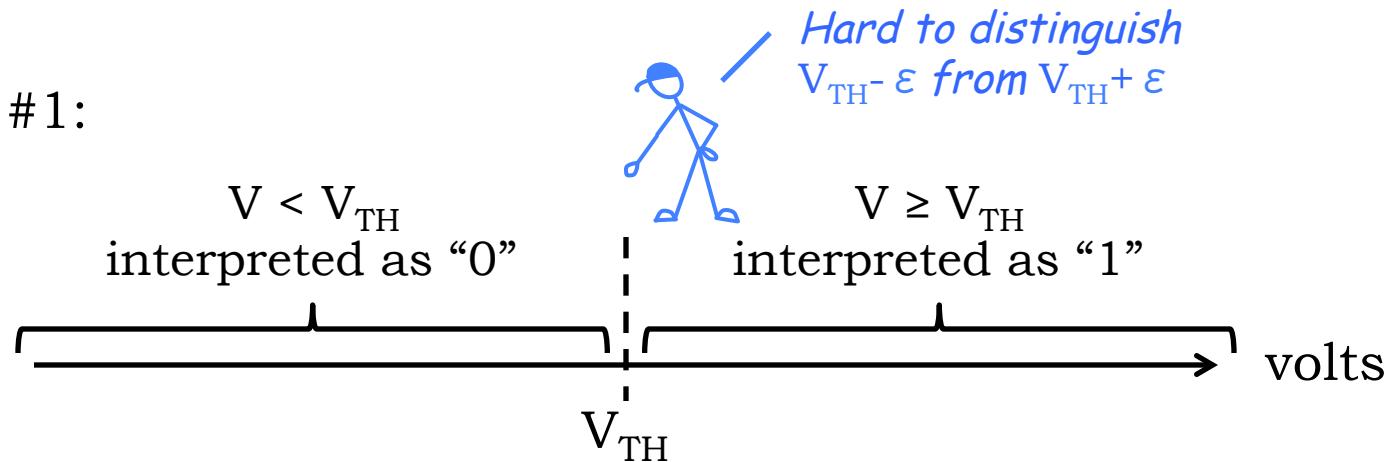
Keep in mind that the world is not digital, we would simply like to engineer it to behave that way. Furthermore, we must use **real physical phenomena** to implement digital designs!

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

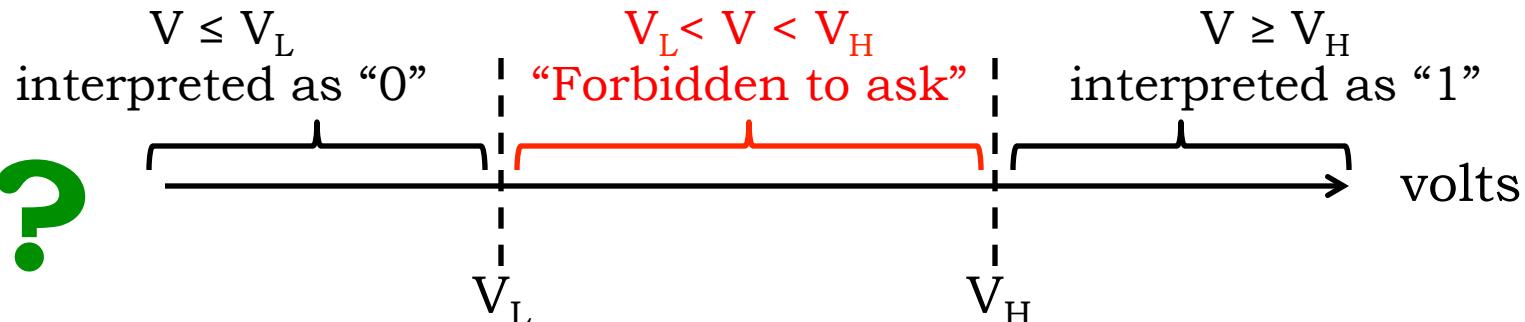
Attempt #1:

✗



Attempt #2:

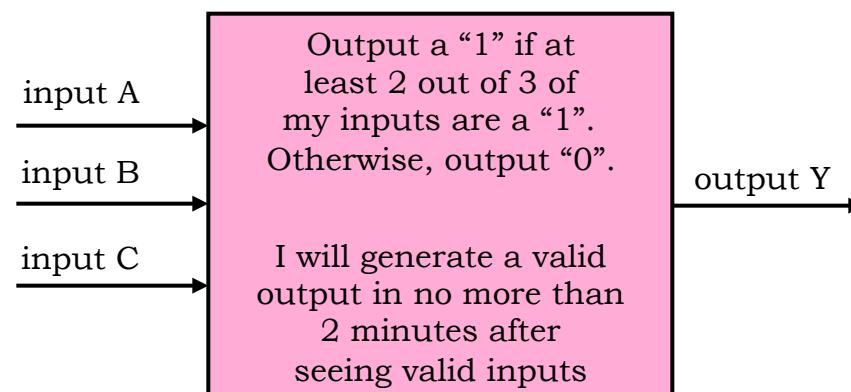
✓ ?



A Digital Processing Element

A combinational device is a circuit element that has

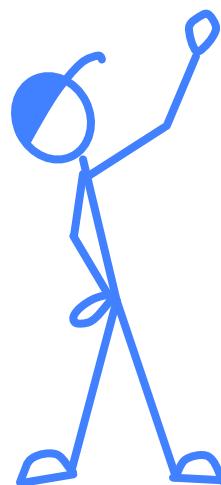
- Static discipline
- one or more digital *inputs*
 - one or more digital *outputs*
 - a *functional specification* that details the value of each output for every possible combination of valid input values
 - a *timing specification* consisting (at minimum) of an upper bound t_{PD} on the required time for the device to compute the specified output values from an arbitrary set of stable, valid input values



A Combinational Digital System

A set of interconnected elements is a combinational device if

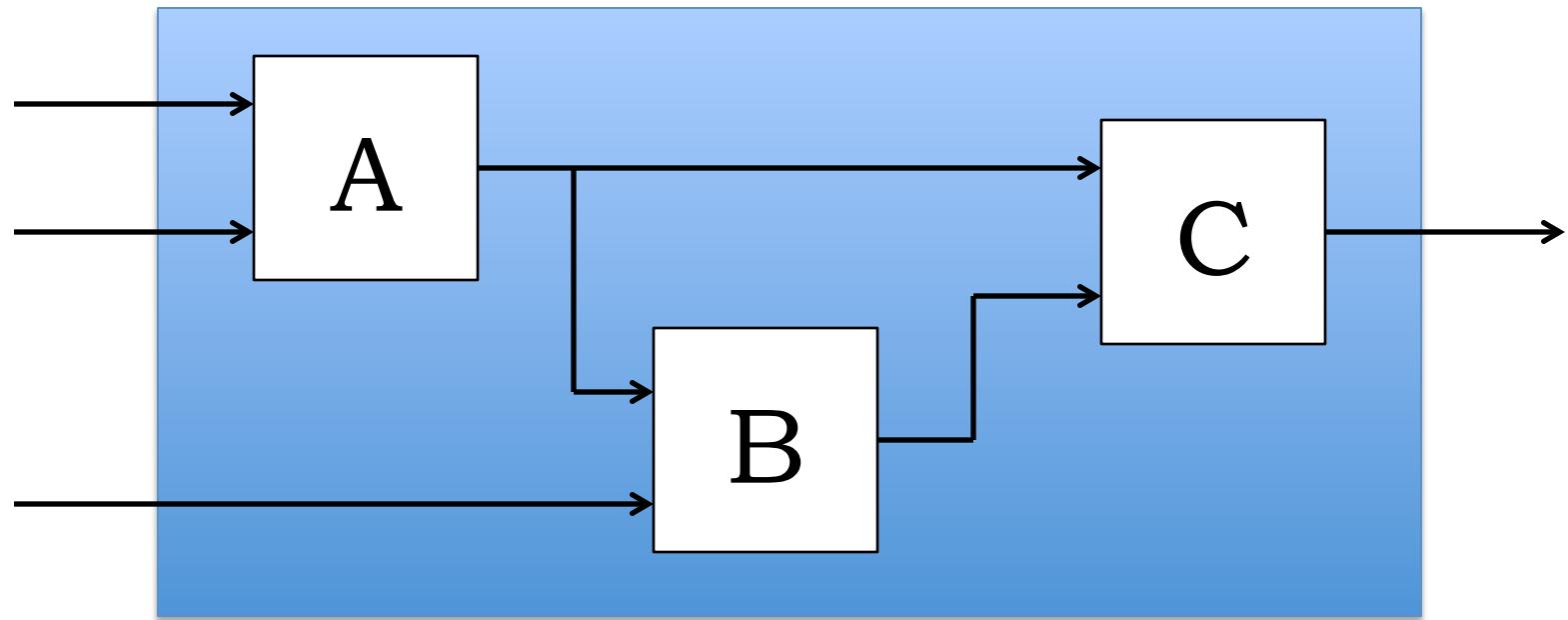
- each circuit element is combinational
- every input is connected to exactly one output or to some vast supply of constant 0's and 1's
- the circuit contains no directed cycles



Why is this true?

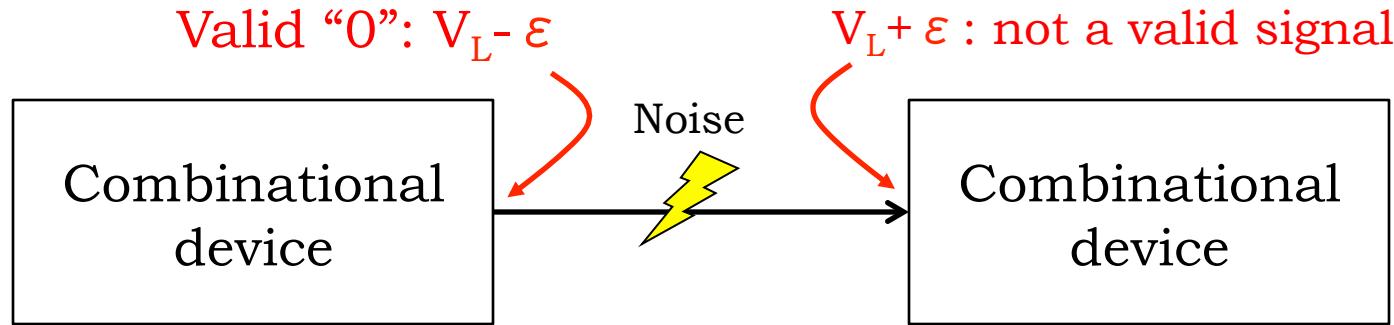
Is This a Combinational Device?

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

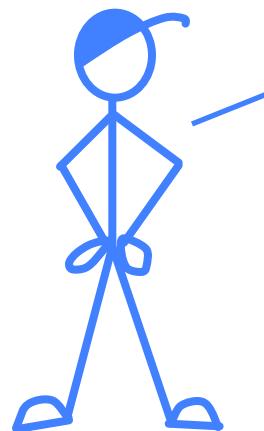


- Does it have digital inputs?
- Does it have digital outputs?
- Can you derive a functional description?
- Can you derive a t_{PD} ?

Will This System Work?



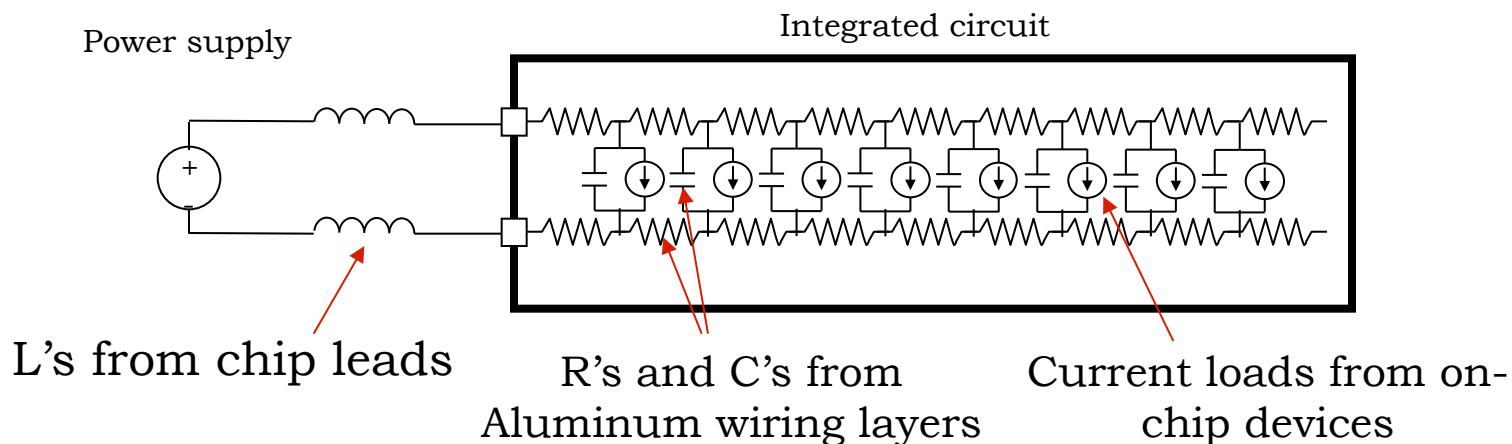
Upstream device transmits a signal at $V_L - \varepsilon$, a valid “0”. Noise on the connecting wire causes the downstream device to receive $V_L + \varepsilon$, a signal in the forbidden zone.



Hmm. Looks like the output voltage needs to be adjusted so that a signal will still be valid when it reaches an input even if there's noise.

Where Does Noise Come From?

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

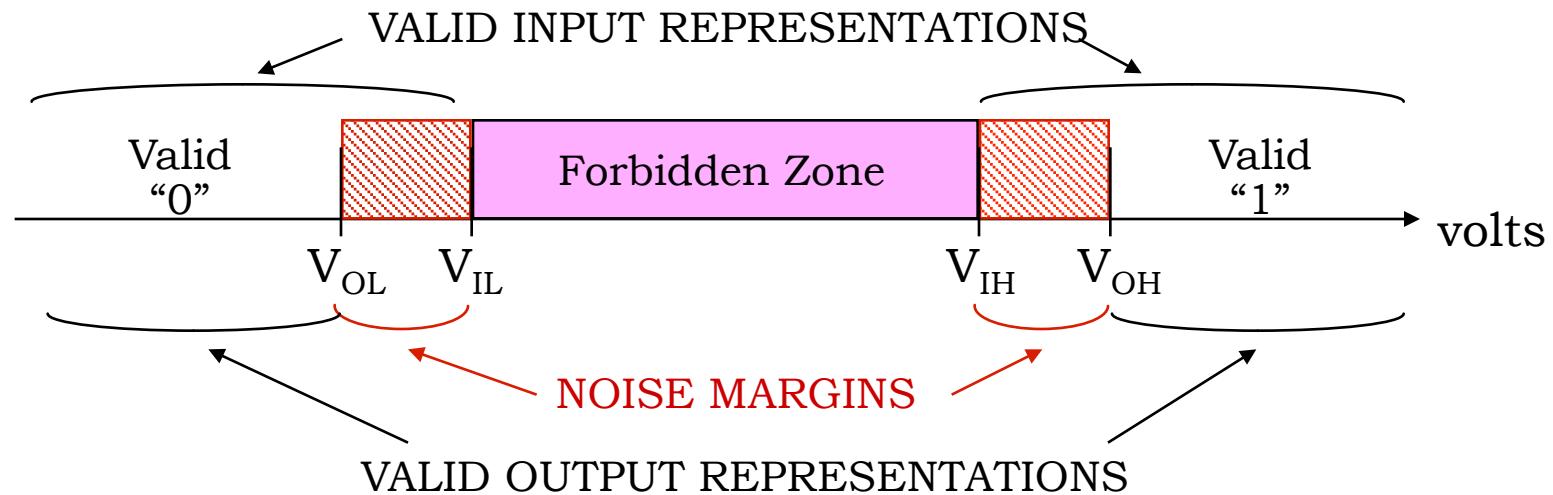


- Imprecision of component values
 - Manufacturing variations, allowable tolerances
- Environmental effects
 - External EM fields, temperature variations, etc.
- ...

Needed: Noise Margins!

Proposed fix: separate specifications for inputs and outputs

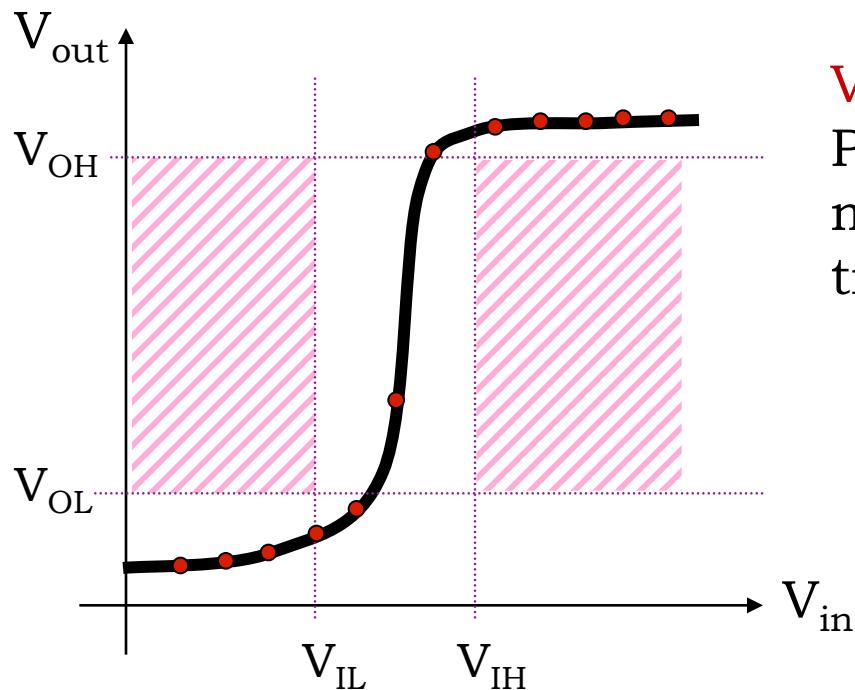
- digital output: “0” $\leq V_{OL}$, “1” $\geq V_{OH}$
- digital input: “0” $\leq V_{IL}$, “1” $\geq V_{IH}$
- $V_{OL} < V_{IL} < V_{IH} < V_{OH}$



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

A Buffer

A simple *combinational device*:

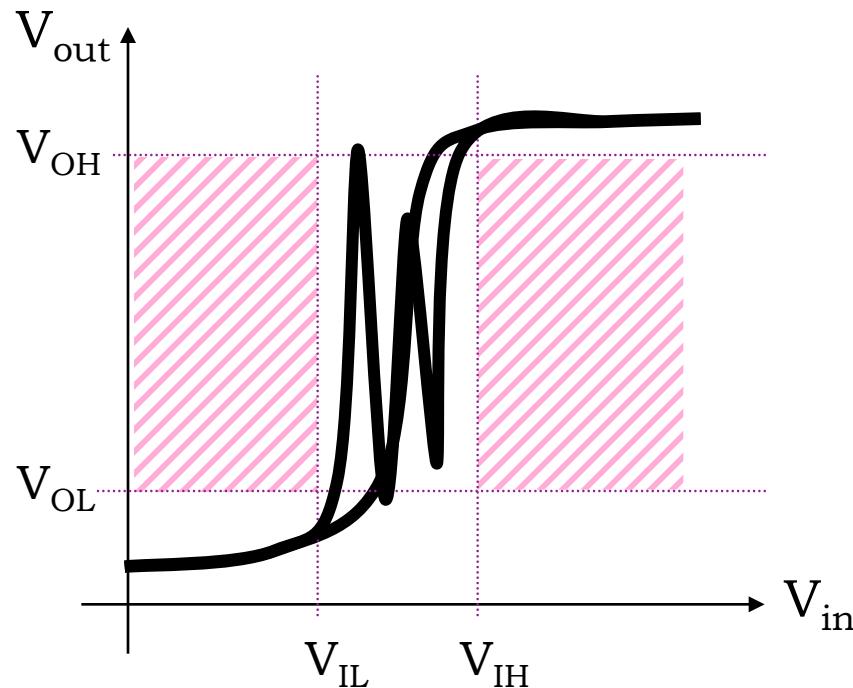


Voltage Transfer Characteristic (VTC):
Plot of V_{out} vs. V_{in} where each measurement is taken after any transients have died out.

Note: VTC does not tell you anything about how fast a device is — it measures static behavior not dynamic behavior

Static Discipline requires that the VTC avoid the shaded regions (aka “*forbidden zones*”), which correspond to *valid* inputs but *invalid* outputs.

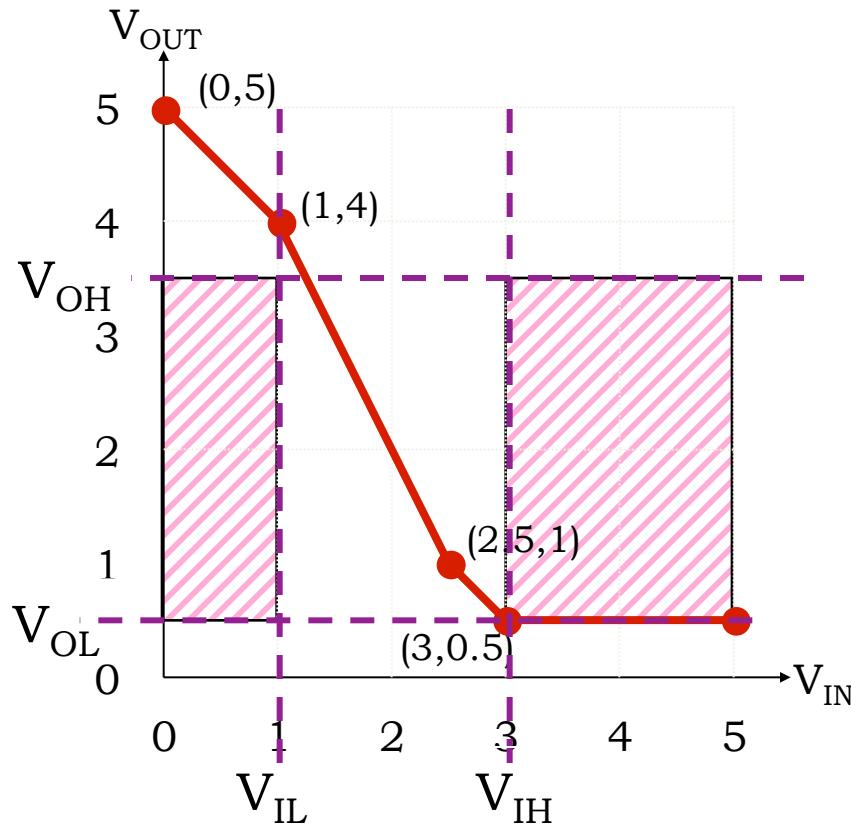
Voltage Transfer Characteristic



- 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**.

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 $V_{IN} \leq V_{IL}$.

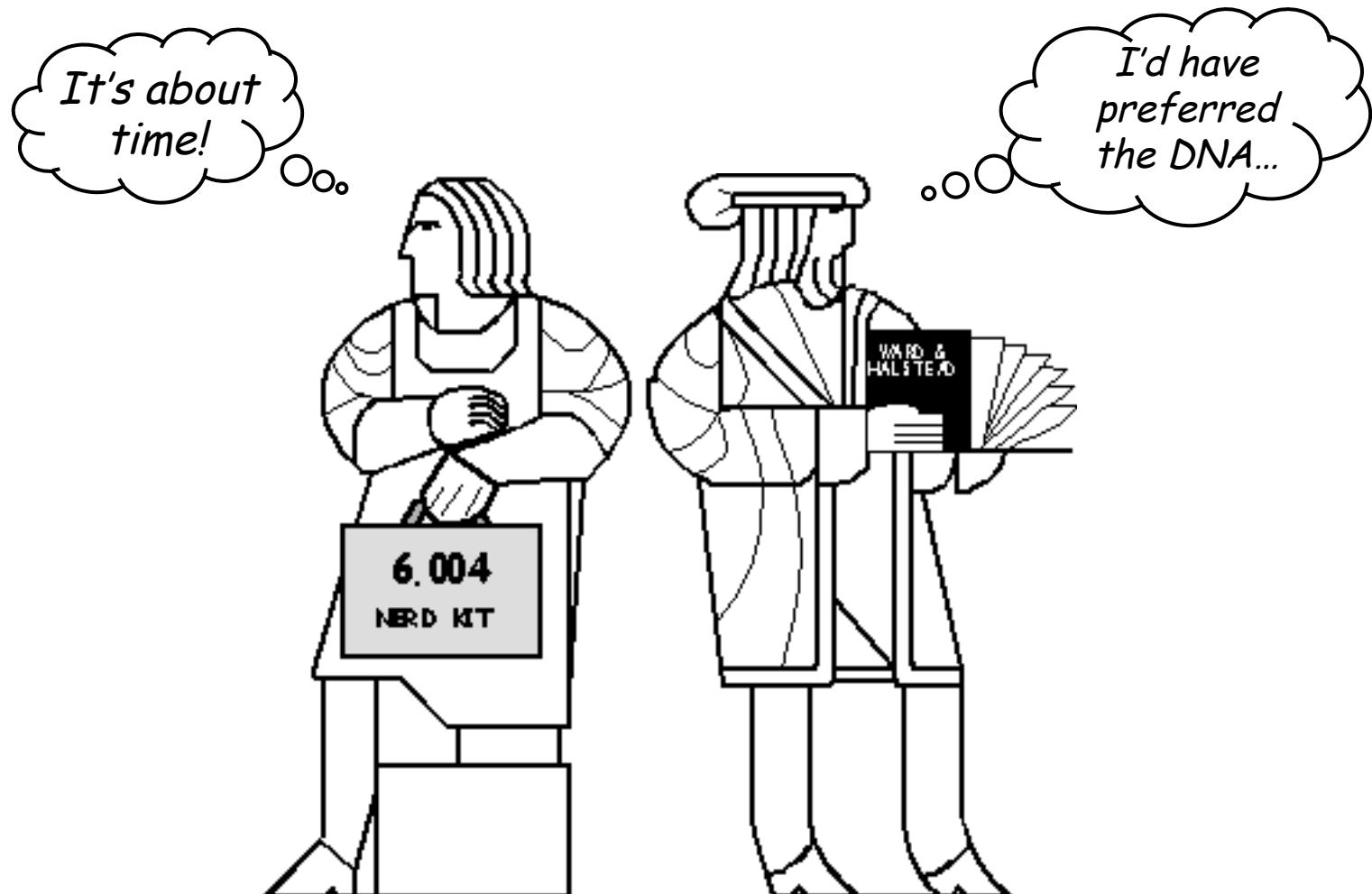
Try $N = 0.5$ V

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

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

Next Time: Building Logic with Transistors

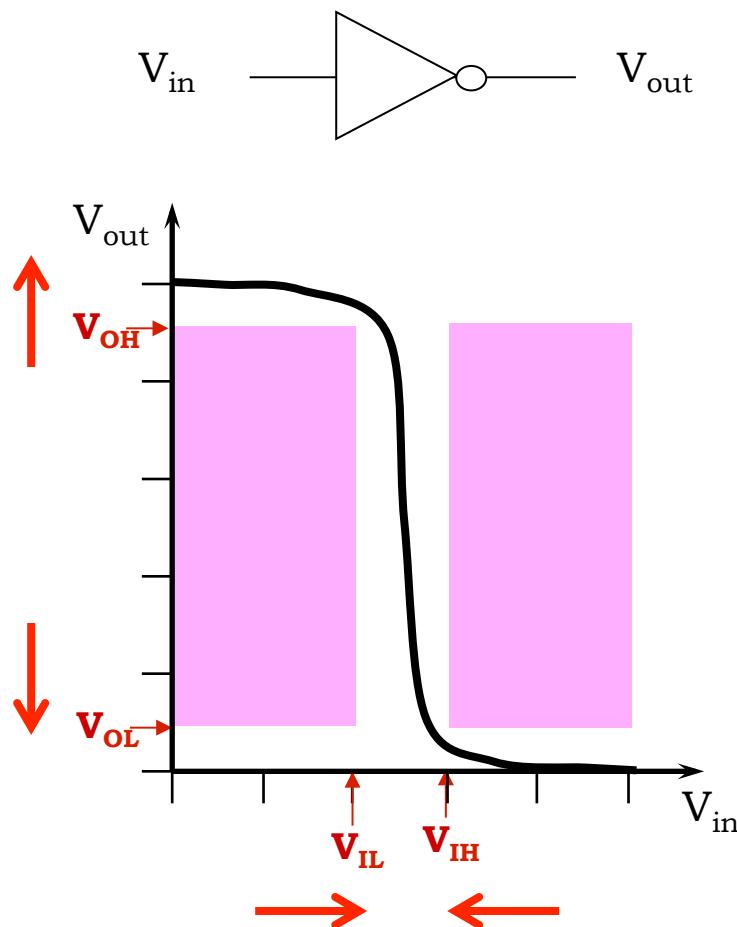


3. CMOS Technology

6.004x Computation Structures
Part 1 – Digital Circuits

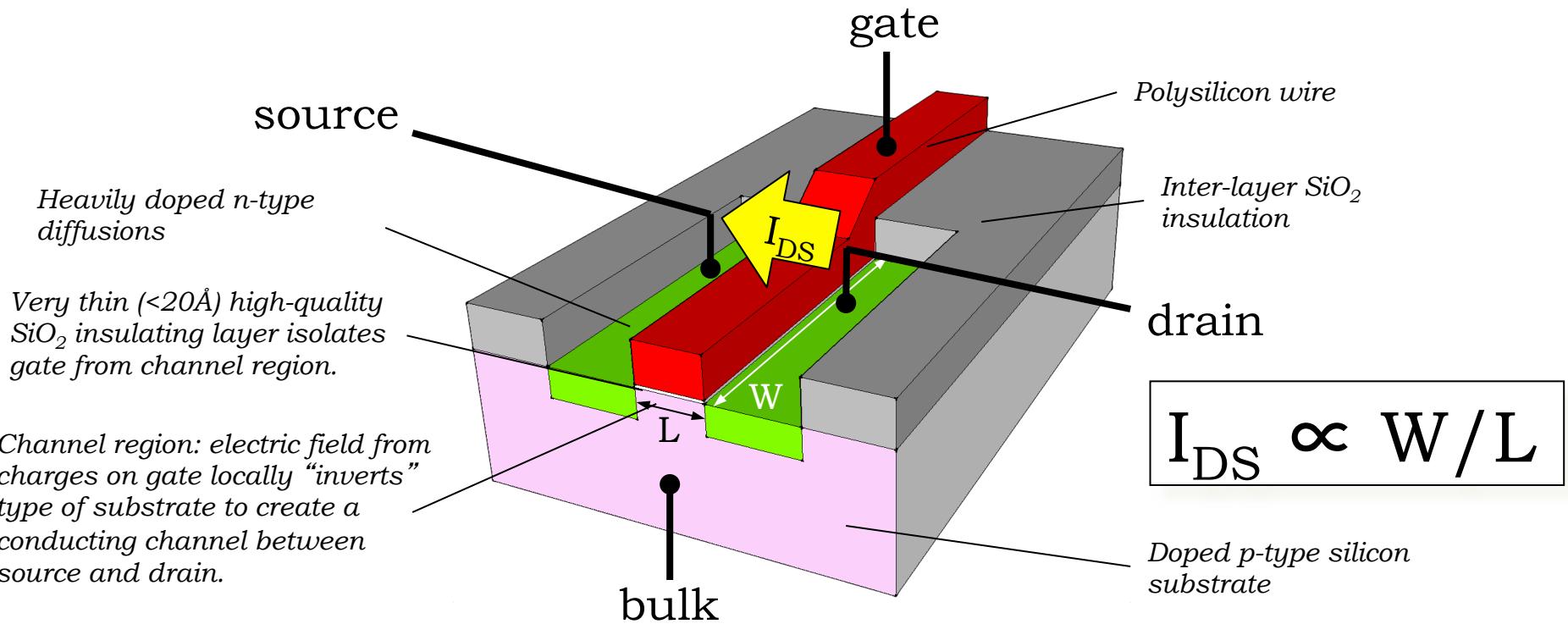
Copyright © 2015 MIT EECS

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?)

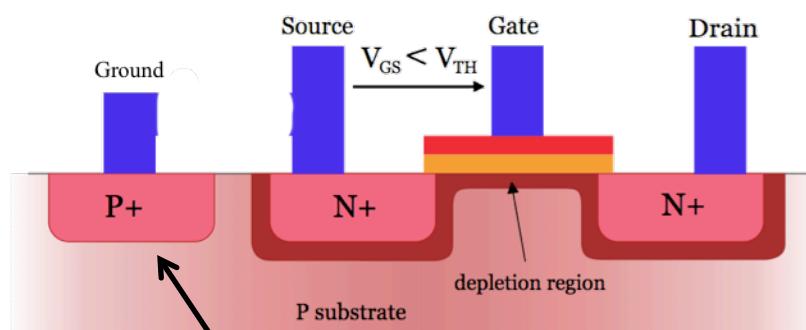
N-Channel MOSFET: Physical View



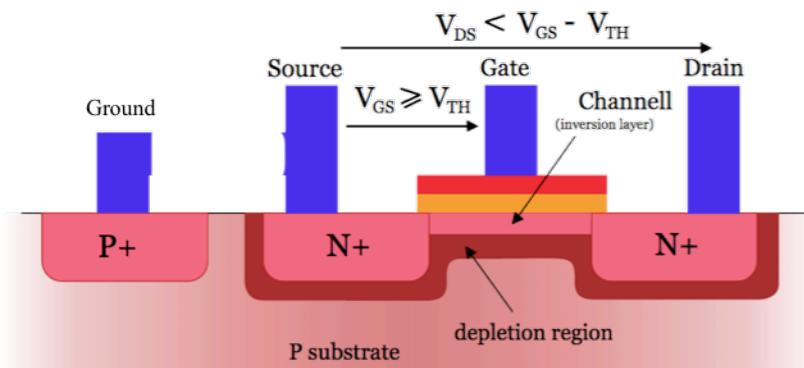
MOSFETs (metal-oxide-semiconductor field-effect transistors) are four-terminal voltage-controlled switches. Current flows between the diffusion terminals if the voltage on the gate terminal is large enough to create a conducting “channel”, otherwise the mosfet is off and the diffusion terminals are not connected.

N-Channel MOSFET: Electrical View

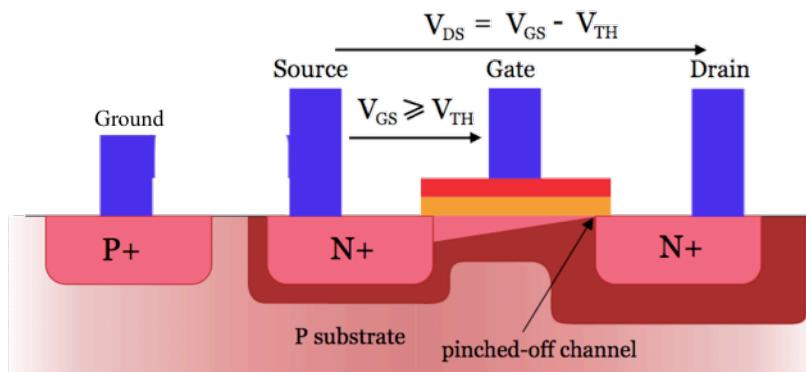
The four terminals of a Field Effect Transistor (gate, source, drain and bulk) connect to conductors that generate a set of electric fields in the channel region which depend on the relative voltages of each terminal.



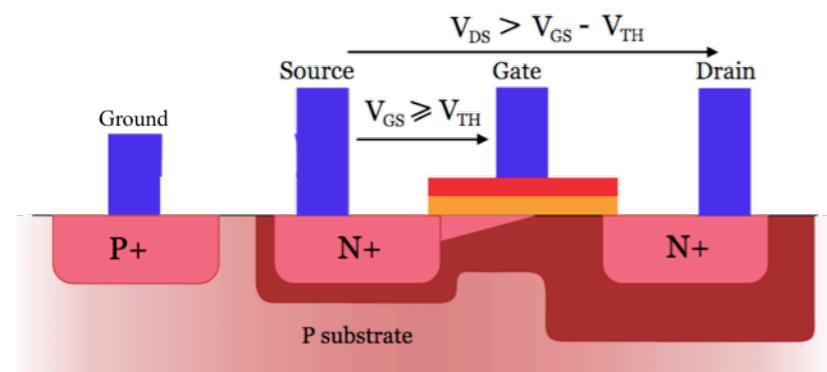
Want $V_P \leq V_N$



Linear operating region (ohmic mode)



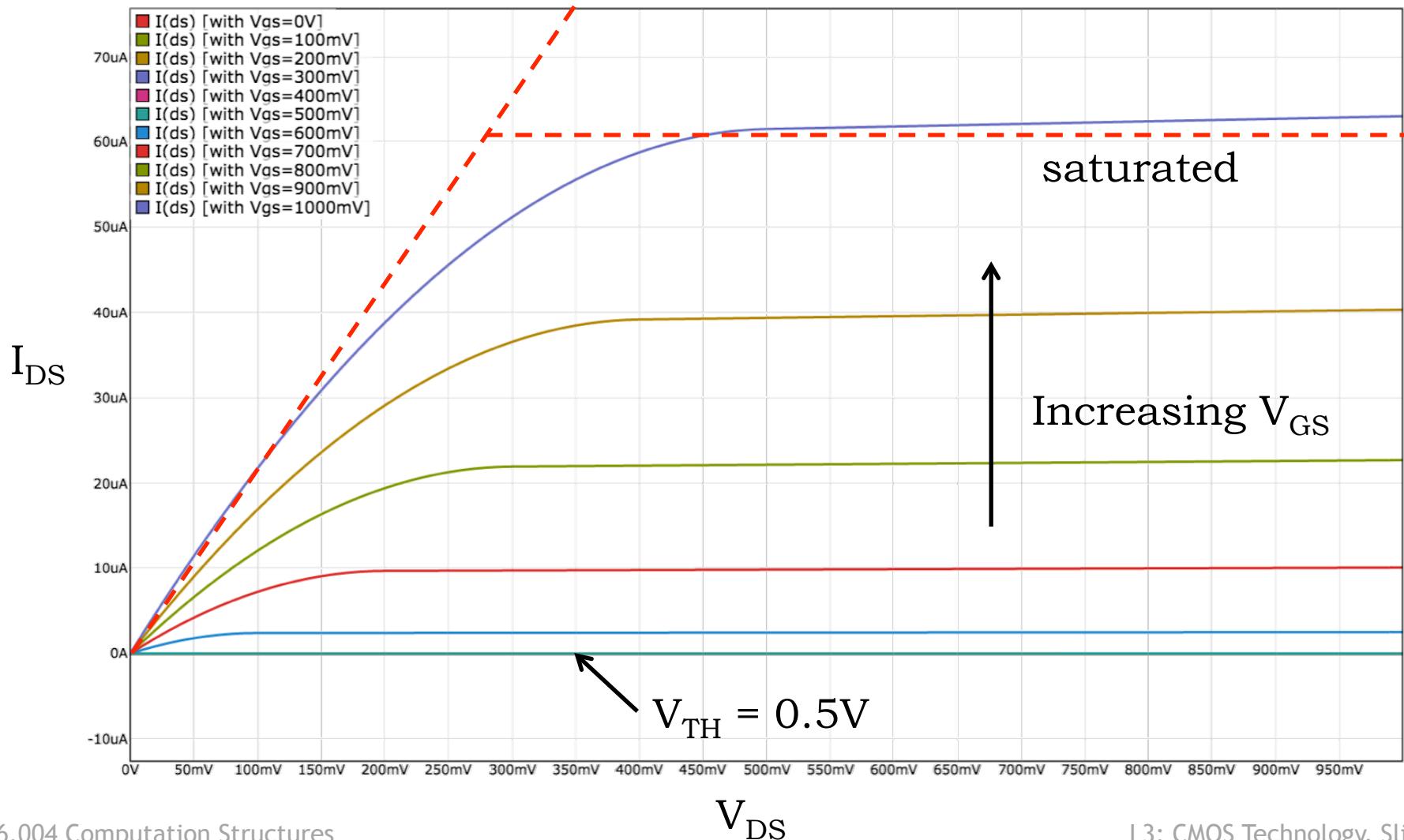
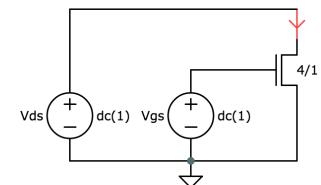
Saturation mode at point of pinch-off



Saturation mode

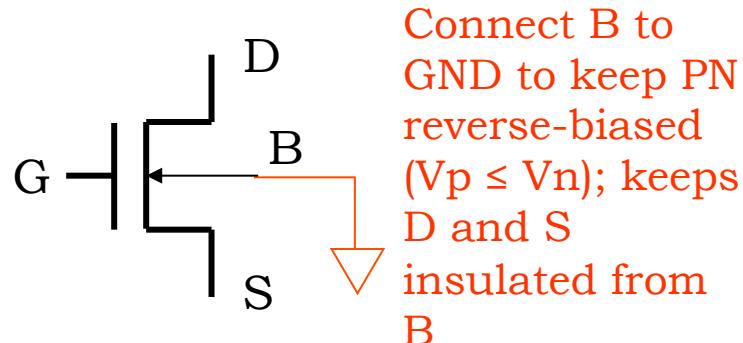
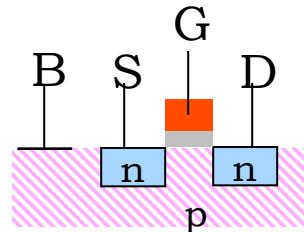
N-channel MOSFET I_{DS} vs. V_{DS}

Ohmic: $I_{DS} = V_{DS}/R$

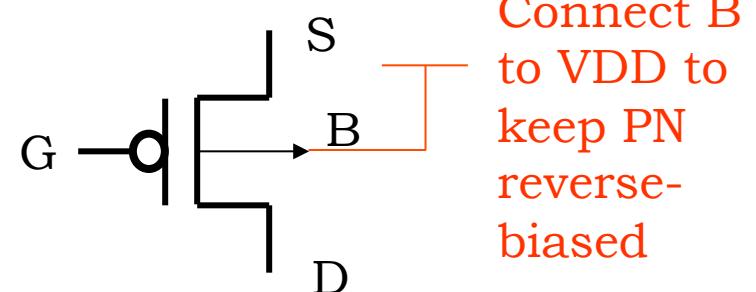
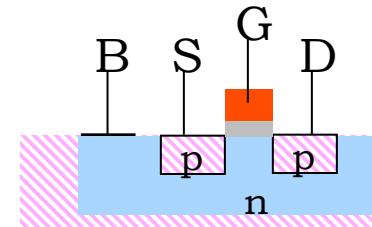


FETs Come in Two Flavors

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.



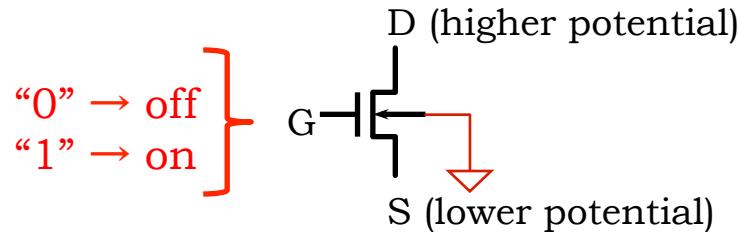
The use of both NFETs and PFETs – complimentary transistor types – is a key to CMOS (complementary MOS) logic families.

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



NFET Operating regions:

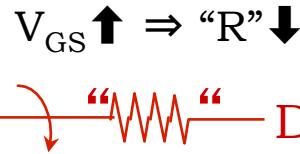
“off”:

$$V_{GS} < V_{TH,NFET}$$

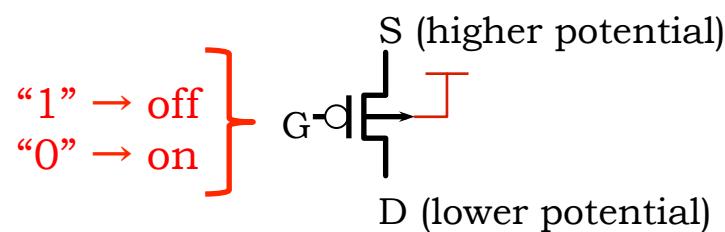


“on”:

$$V_{GS} > V_{TH,NFET}$$



NFET threshold = ~0.5V



PFET Operating regions:

“off”:

$$V_{GS} > V_{TH,PFET}$$

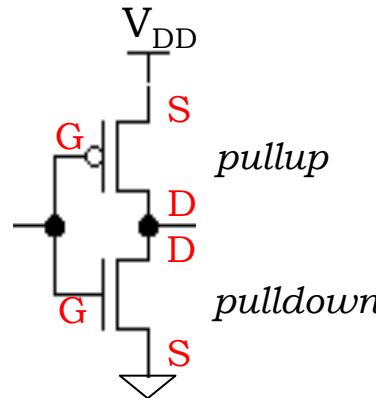


“on”:

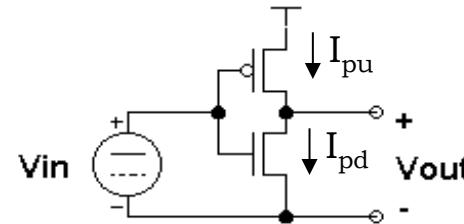
$$V_{GS} < V_{TH,PFET}$$



PFET threshold = ~ -0.5V

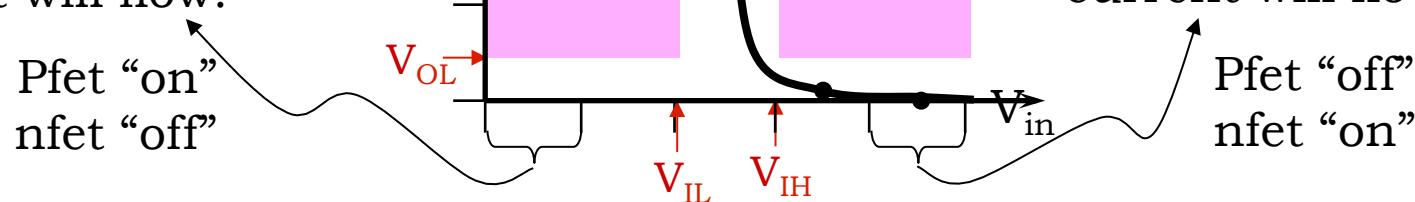


CMOS Inverter VTC



Steady state reached when V_{out} reaches value where $I_{pu} = I_{pd}$.

When V_{IN} is low, the nfet is off and the pfet is on, so current flows into the output node and V_{OUT} eventually reaches V_{DD} ($> V_{OH}$) at which point no more current will flow.



When V_{IN} is high, the pfet is off and the nfet is on, so current flows out of the output node and V_{OUT} eventually reaches GND ($< V_{OL}$) at which point no more current will flow.

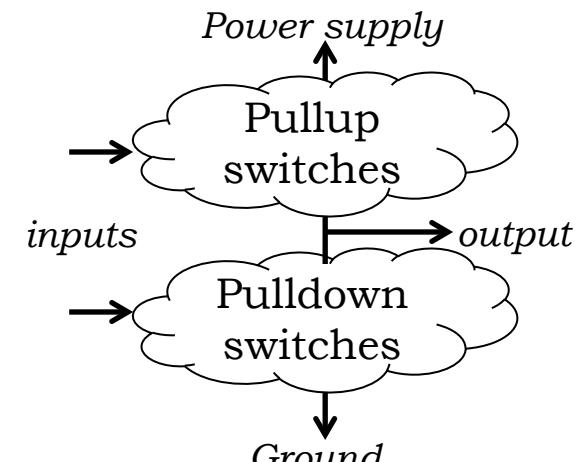
When V_{IN} is in the middle, both the pfet and nfet are “on” and the shape of the VTC depends on the details of the devices’ characteristics. CMOS gates have very high gain in this region (small changes in V_{IN} produce large changes in V_{OUT}) and the VTC is almost a step function.

Beyond Inverters: Complementary pullups and pulldowns

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

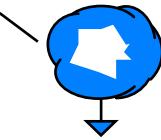
We want *complementary* pullup and pulldown logic, i.e., the pulldown should be “on” when the pullup is “off” and vice versa.

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

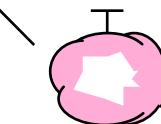


Since there's plenty of capacitance on the output node, when the output becomes disconnected it “remembers” its previous voltage -- at least for a while. The “memory” is the load capacitor's charge. Leakage currents will cause eventual decay of the charge (that's why DRAMs need to be refreshed!).

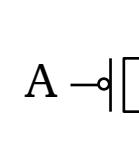
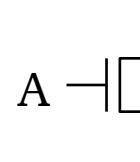
What a nice
 V_{OH} you have...



Thanks. It runs
in the family...

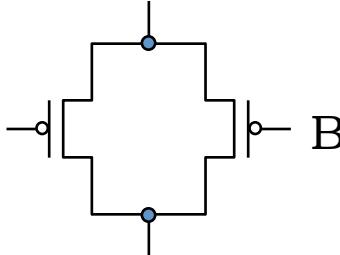
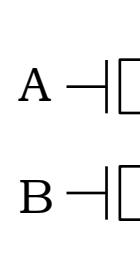


CMOS Complements



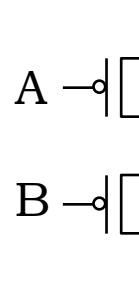
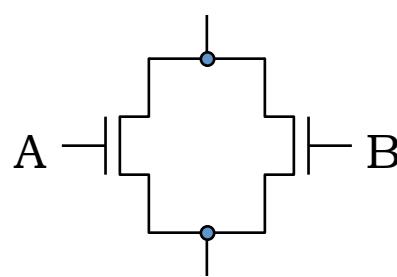
conducts when A is high

conducts when A is low: \bar{A}



conducts when A is high
and B is high: $A \cdot B$

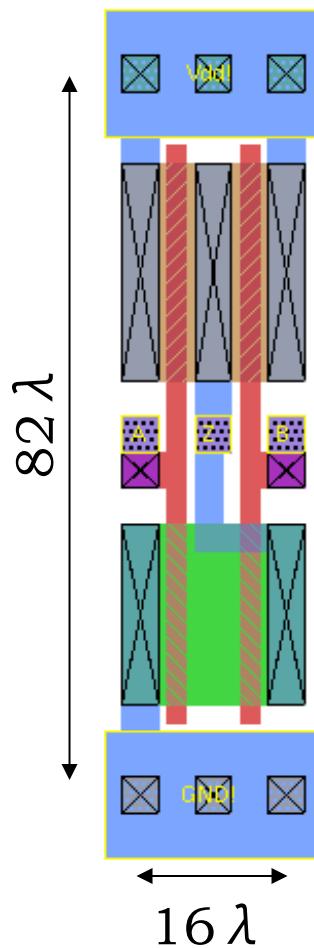
conducts when A is low
or B is low: $A + \bar{B} = A \cdot \bar{B}$



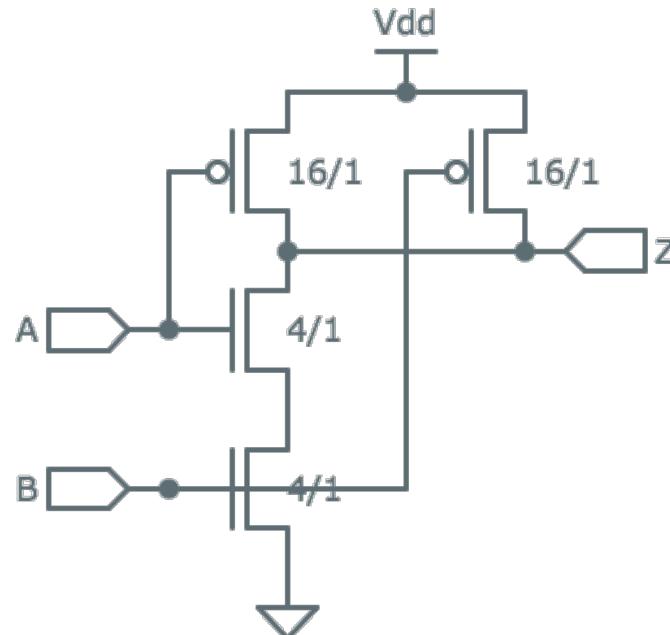
conducts when A is high
or B is high: $A + B$

conducts when \bar{A} is low
and B is low: $\bar{A} \cdot \bar{B} = A + B$

A Pop Quiz!



Current technology: $\lambda = 14\text{nm}$



What function does this gate compute?

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

NAND

COST for an older 45nm process:

- \$3500 per 300mm wafer
- 300mm round wafer = $\pi(150\text{e-}3)^2 = .07\text{m}^2$
- NAND gate = $(82)(16)(45\text{e-}9)^2 = 2.66\text{e-}12\text{m}^2$
- $2.6\text{e}10$ NAND gates/wafer (= 100 billion FETS!)
- marginal cost of NAND gate: **132n\$**

General CMOS Gate Recipe

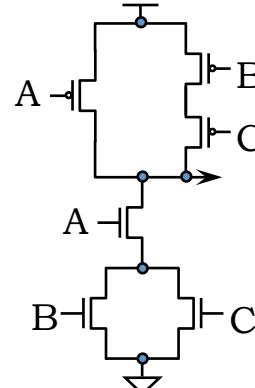
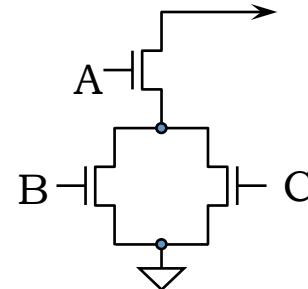
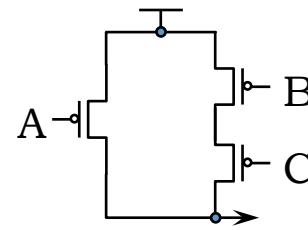
Step 1. Figure out the pullup network that does what you want, e.g.,

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

(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

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



Does this
recipe work
for all logic
functions?



CMOS Gates Are Naturally Inverting

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

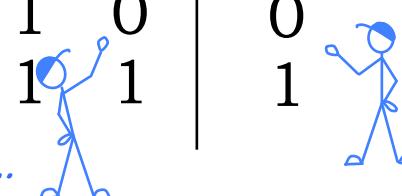
- NFETs go from “off” to “on”
 - pulldown paths connected
 - output may be connected to ground
- PFETs go from “on” to “off”
 - pullup paths disconnected
 - output may be disconnected from V_{DD}

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

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

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

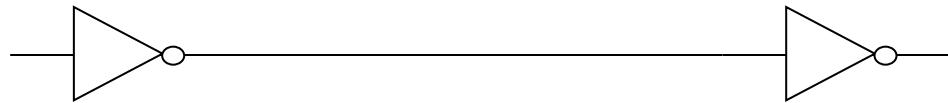
A=1, B rising...



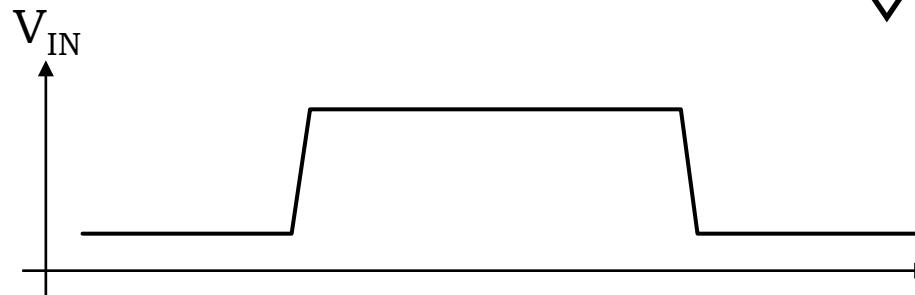
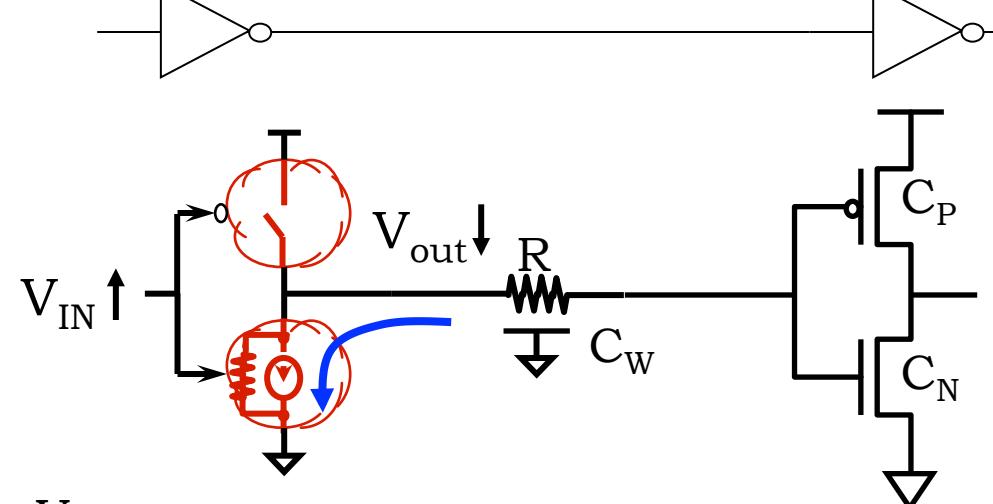
Oops, output is also rising!

CMOS Timing Specifications

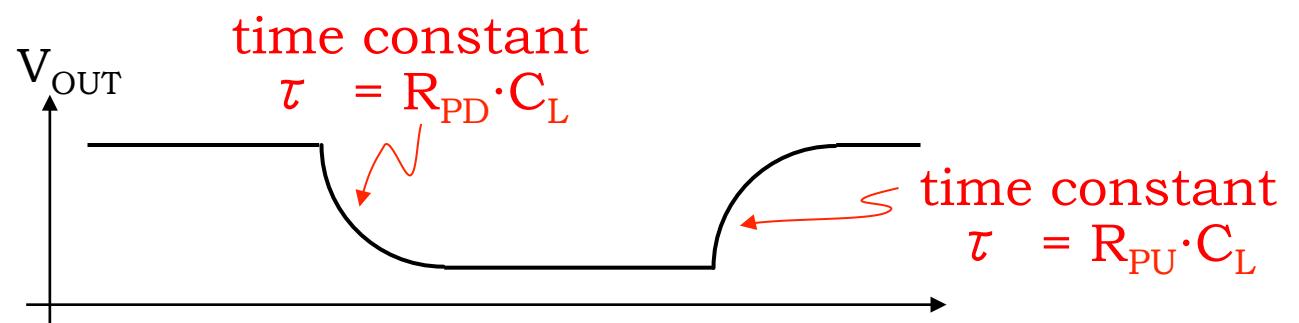
Circuit:



Electrical model:

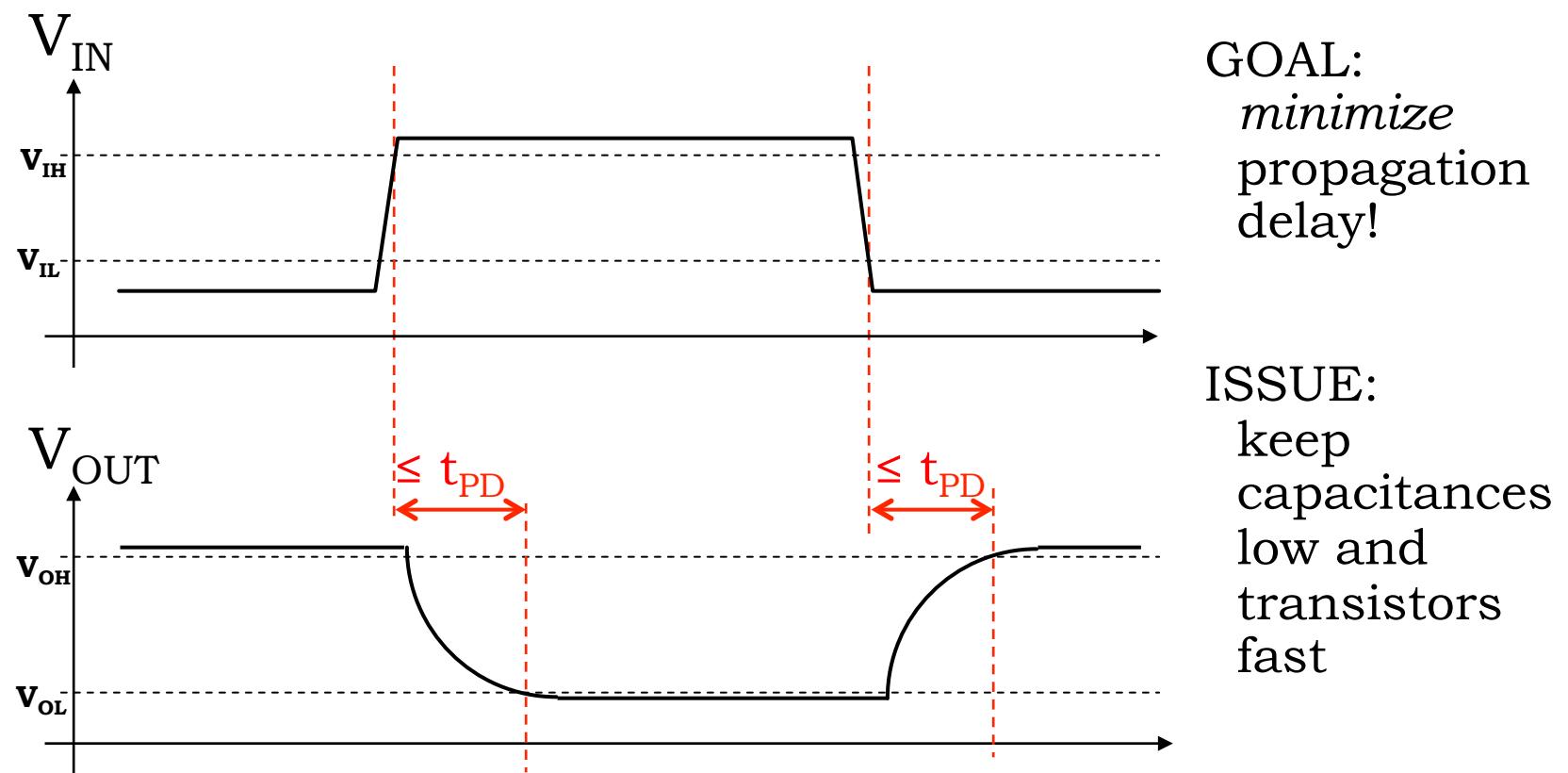


Waveforms:



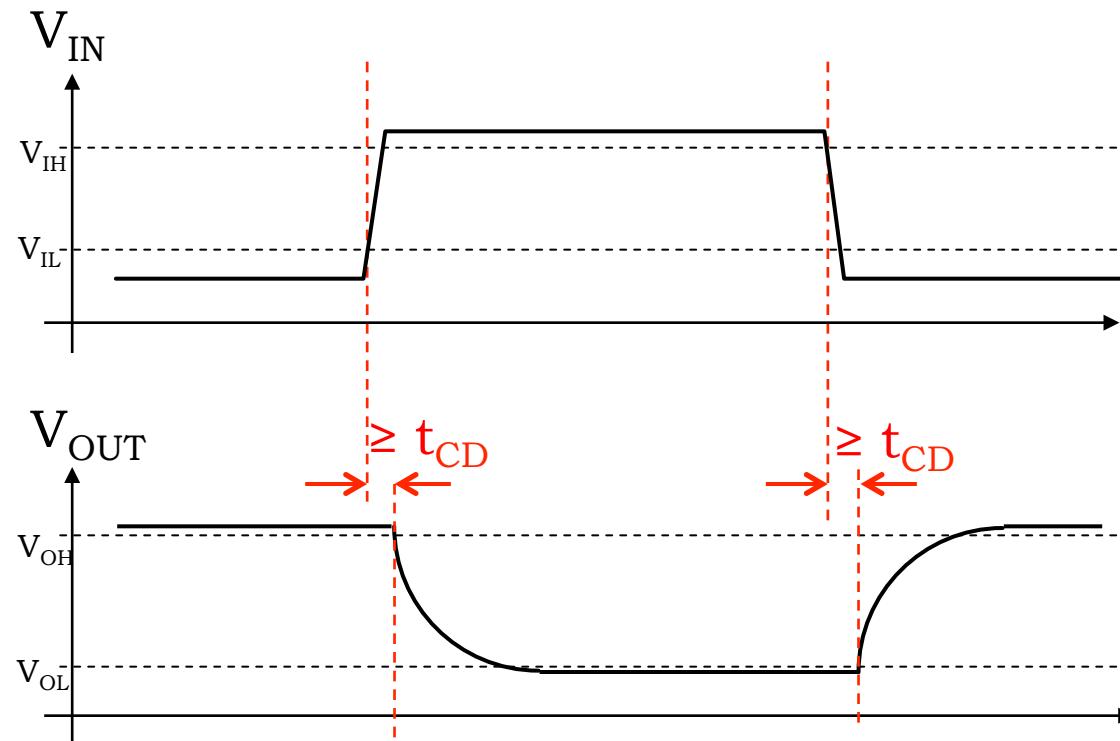
Propagation Delay

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



Contamination Delay

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

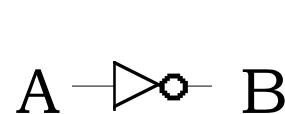


Do we really need t_{CD} ?

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

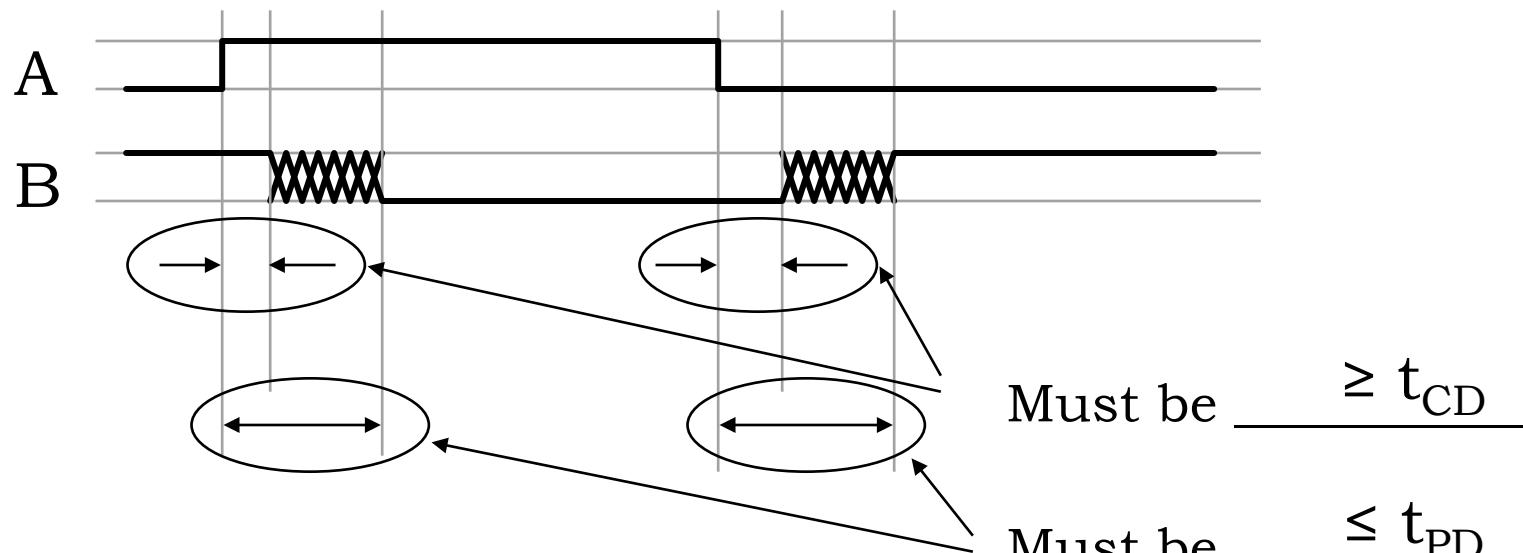
If t_{CD} is not specified, safe to assume it's 0.

The Combinational Contract



A	B
0	1
1	0

t_{PD} propagation delay
 t_{CD} contamination delay



Notes:

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

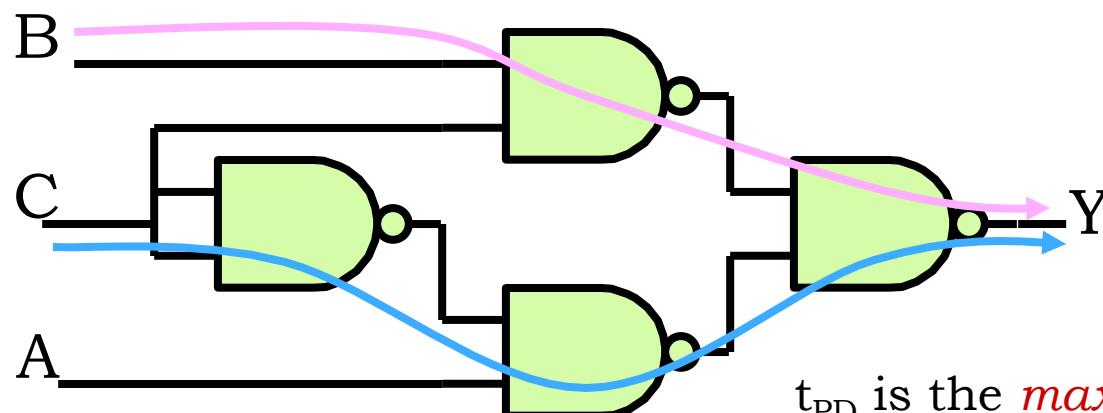
Acyclic Combinational Circuits

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

t_{CD} is the *minimum* cumulative contamination delay over all paths from inputs to outputs

$$t_{PD} = \underline{12} \text{ nS}$$

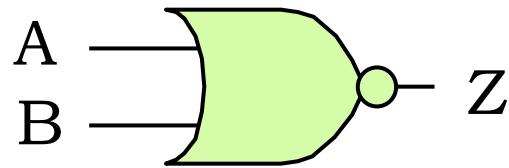
$$t_{CD} = \underline{2} \text{ nS}$$



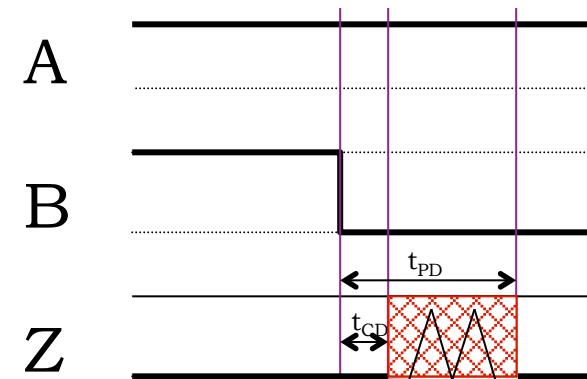
t_{PD} is the *maximum* cumulative propagation delay over all paths from inputs to outputs

One Last Timing Issue...

NOR:



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

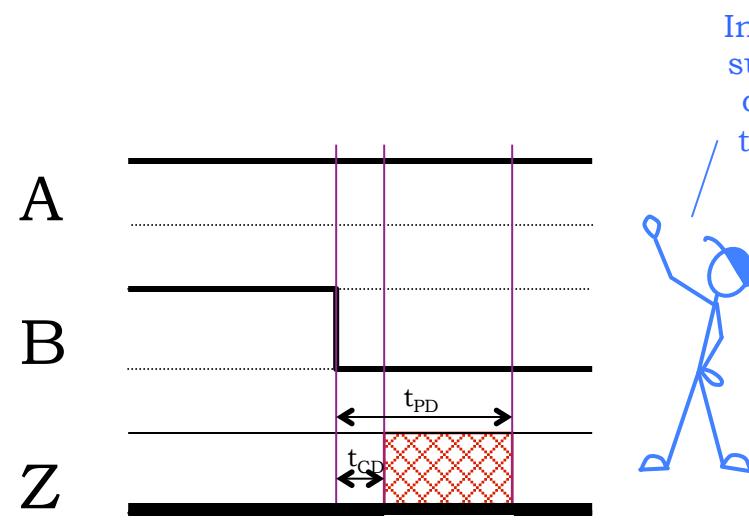
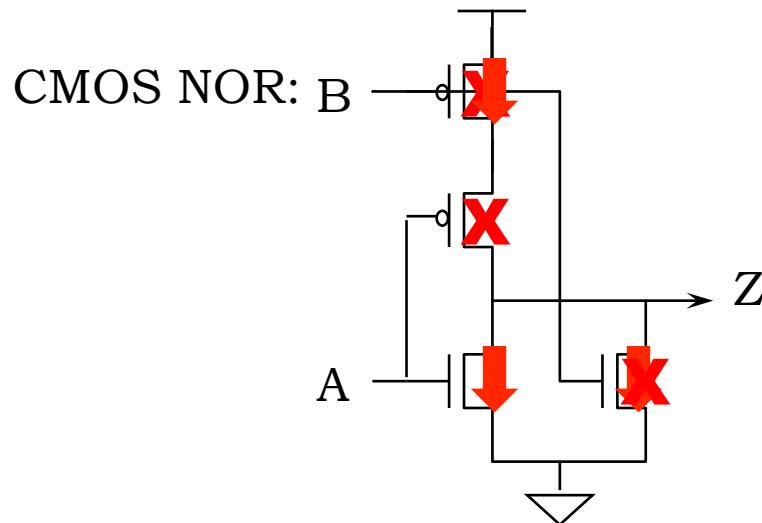


Recall the rules for *combinational devices*:

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.

What Happens In This Case?



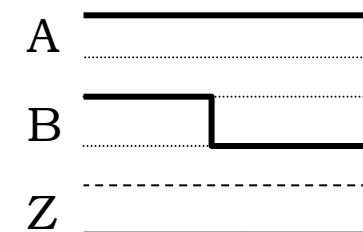
Input A=1 is sufficient to determine the output

LENIENT Combinational Device:

Output guaranteed to be valid when any combination of inputs sufficient to determine the output value has been valid for at least t_{PD} . *Tolerates transitions -- and invalid levels -- on irrelevant inputs!*

NOR:	A	B	Z
	0	0	1
	0	1	0
	1	0	0
	1	1	0

Lenient NOR:	A	B	Z
	0	0	1
	X	1	0
	1	X	0



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

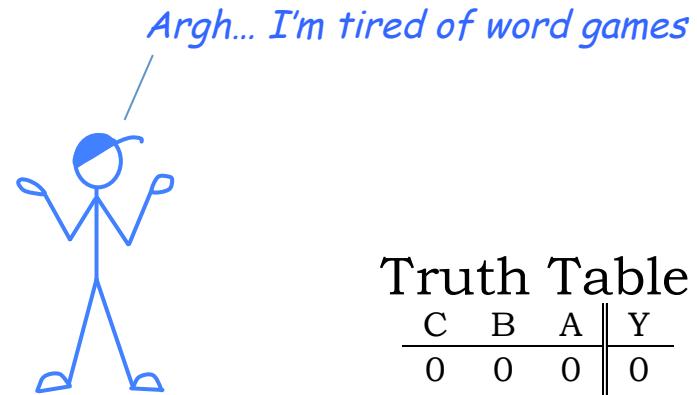
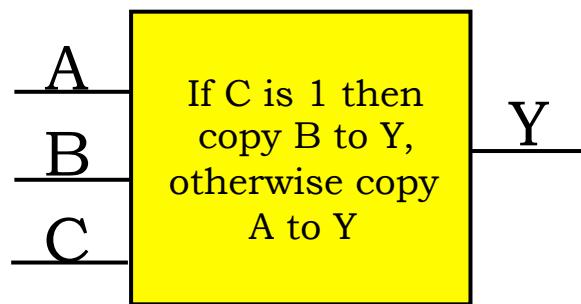
4. Combinational Logic

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS

Functional Specifications

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



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

Concise alternatives:

- *truth tables* are a concise description of the combinational system's function.
- *Boolean expressions* form an algebra whose operations are AND (multiplication), OR (addition), and inversion (overbar).

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

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

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:

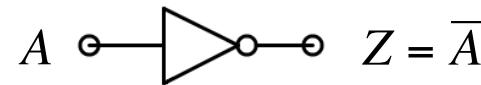
$$Y = \bar{C}\bar{B}A + \bar{C}BA + C\bar{B}\bar{A} + CBA$$

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

Sum-of-products Building Blocks

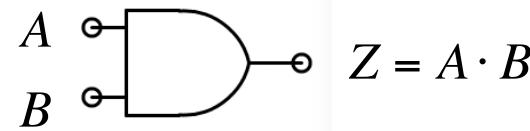
INVERTER:



$$Z = \overline{A}$$

A	Z
0	1
1	0

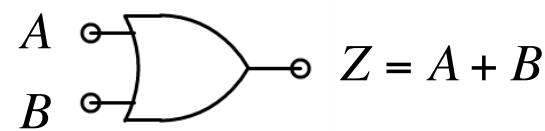
AND:



$$Z = A \cdot B$$

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

OR:



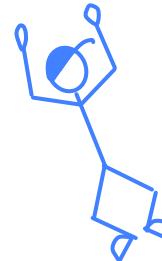
$$Z = A + B$$

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

Straightforward Synthesis

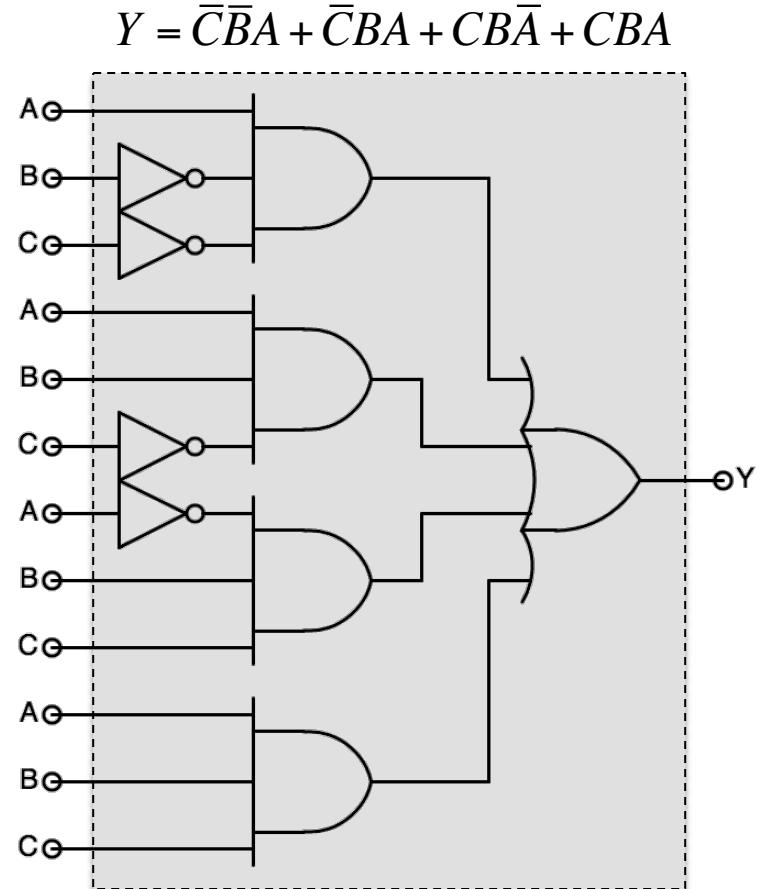
We can implement
SUM-OF-PRODUCTS
with just three levels of
logic:

1. Inverters
2. ANDs
3. OR



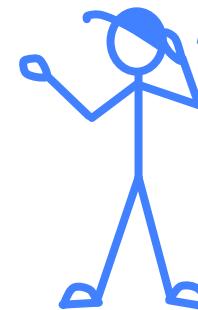
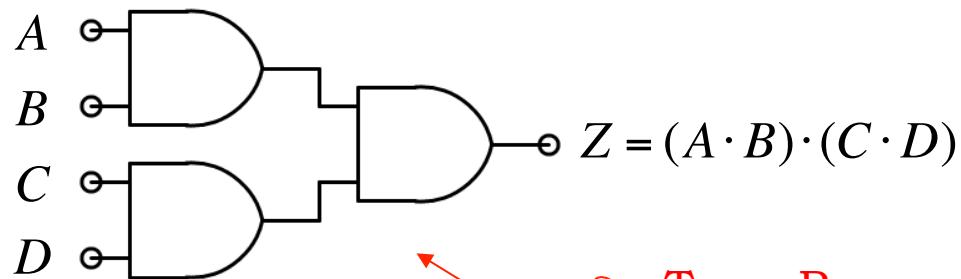
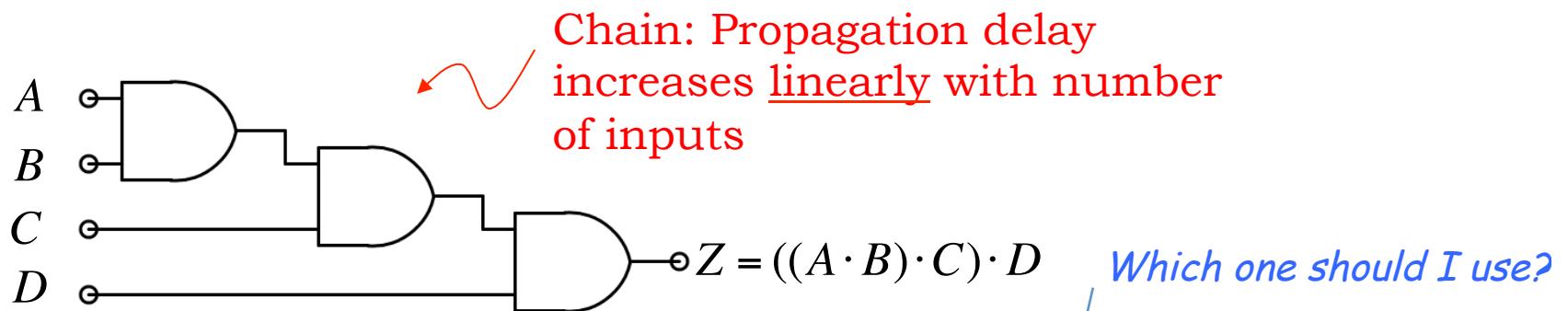
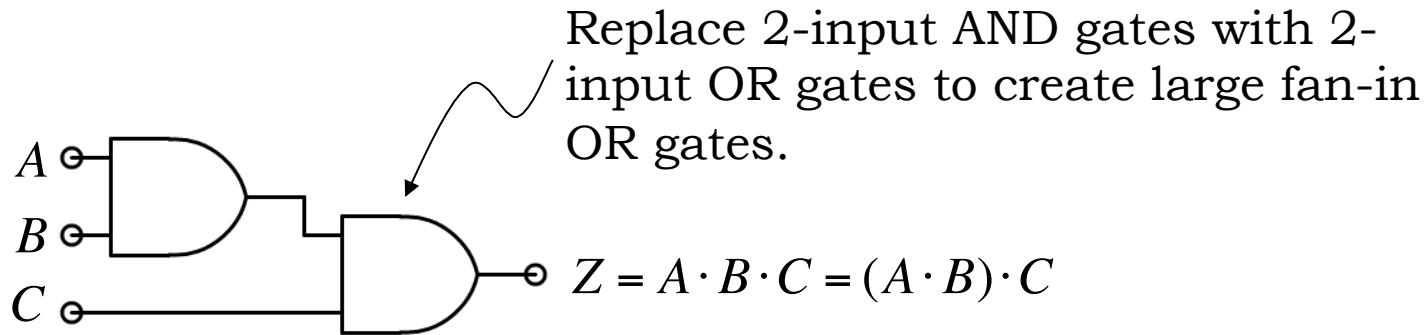
*-it's systematic!
-it works!
-it's easy!
-are we done yet???*

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!

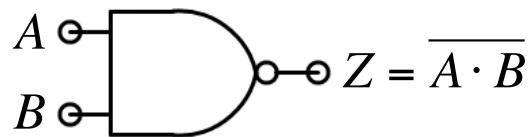
ANDs and ORs with > 2 Inputs



Tree: Propagation delay increases logarithmically with number of inputs

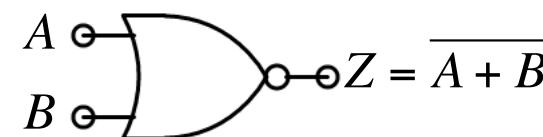
More Building Blocks

NAND (not AND)



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

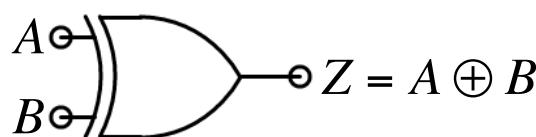
NOR (not OR)



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)



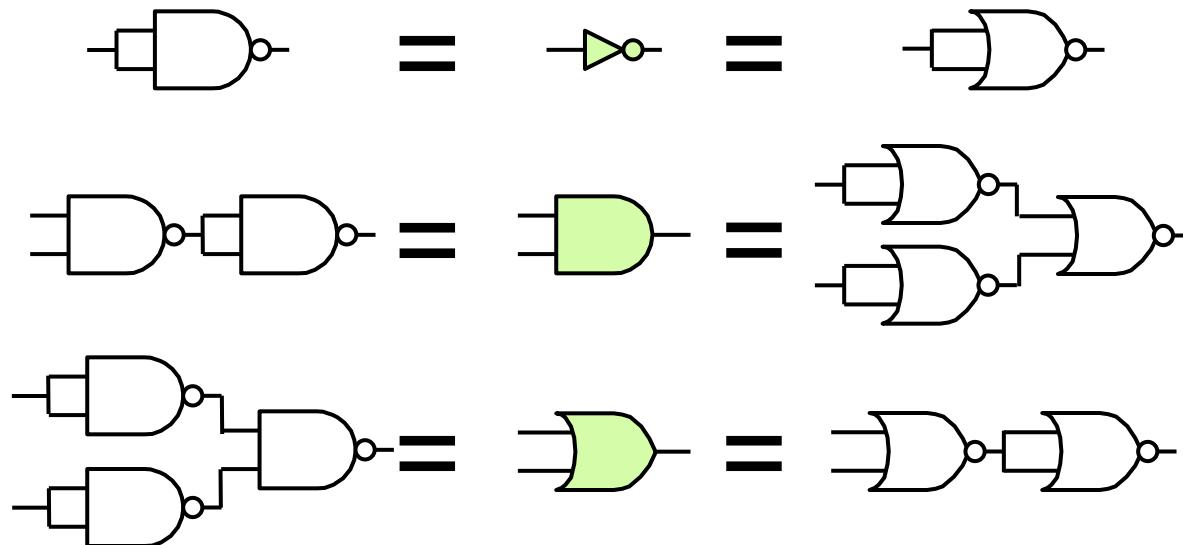
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

XOR is very useful when implementing parity and arithmetic logic. Also used as a “programmable inverter”: if $A=0$, $Z=B$; if $A=1$, $Z=\sim B$

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

Universal Building Blocks

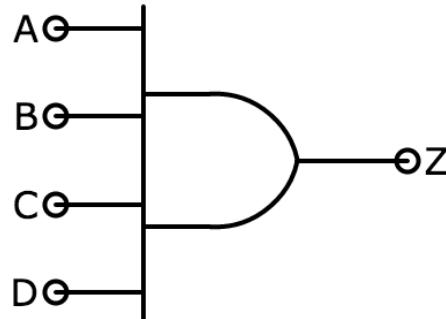
NANDs and NORs are universal:



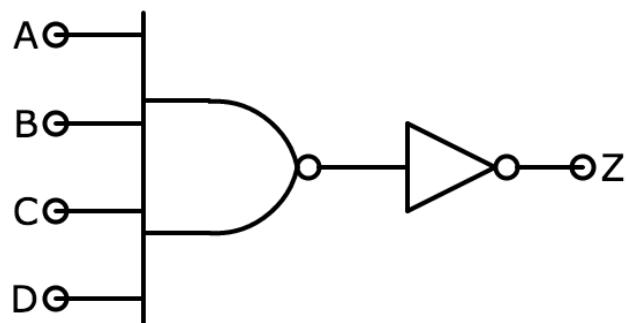
Any logic function can be implemented using only NANDs (or, equivalently, NORs). Good news for CMOS technologies!

CMOS ❤️ Inverting Logic

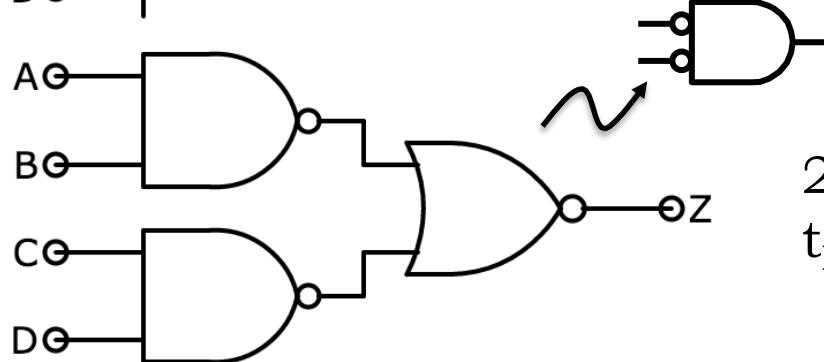
See “The Standard Cell Library” handout in *Updates & Handouts*



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



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



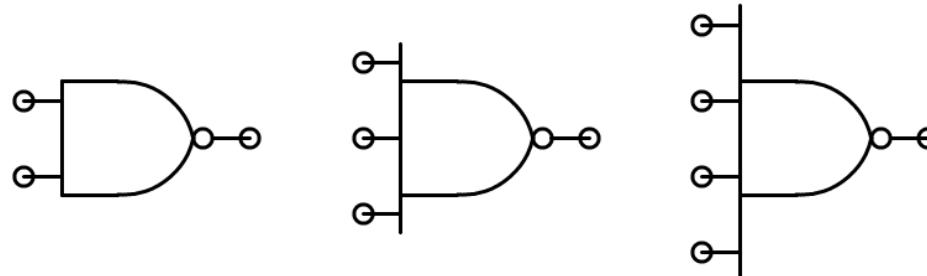
2*NAND2 + NOR2:
 $t_{PD} = 80 \text{ ps}$, size = $30 \mu^2$

Demorgan's Laws:

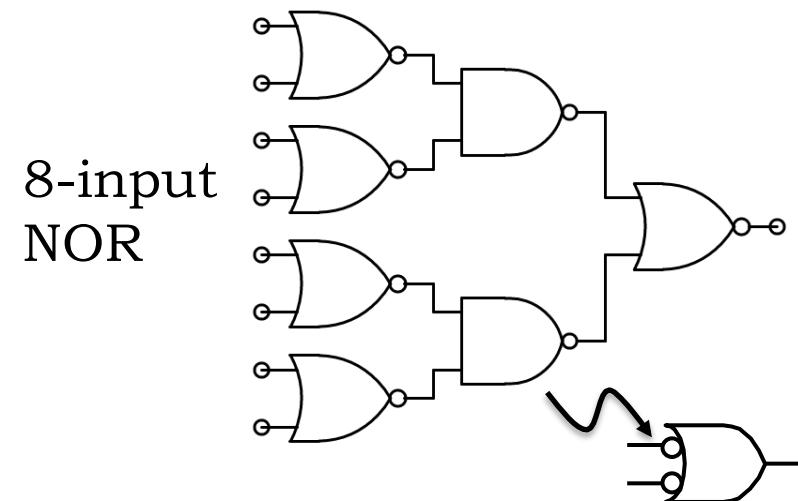
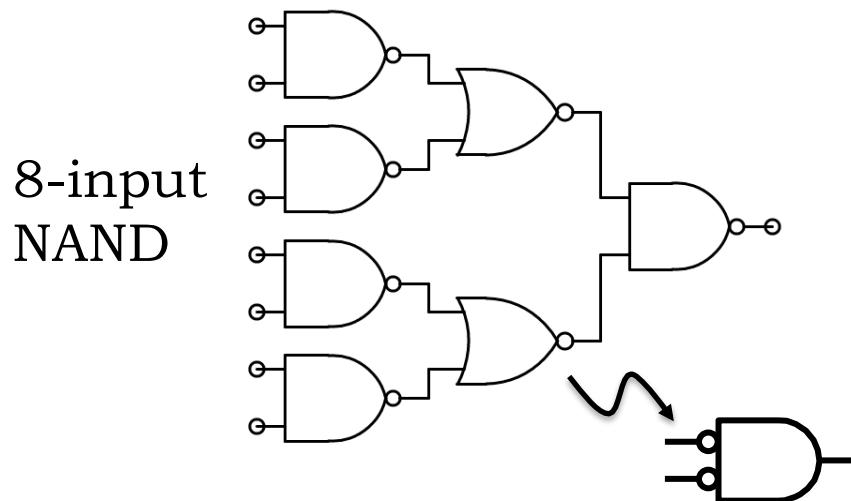
$$\overline{\overline{A} \cdot \overline{B}} = \overline{A + B}$$
$$\overline{\overline{A} + \overline{B}} = \overline{A \cdot B}$$

Wide NANDs and NORs

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

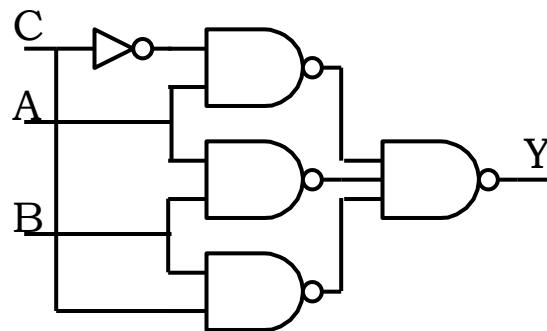


But for a large number of inputs, the series connections of too many MOSFETs can lead to very large effective R.
Design note: use trees of smaller devices...



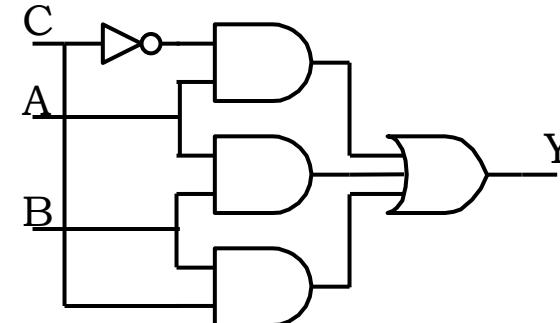
CMOS Sum-of-products Implementation

NAND-NAND

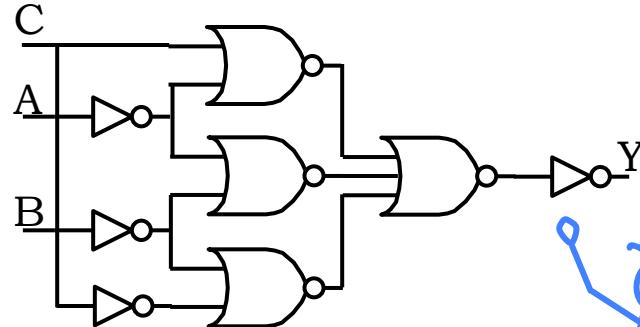


$$\overline{AB} = \overline{\overline{A}} + \overline{\overline{B}}$$

“Pushing Bubbles”

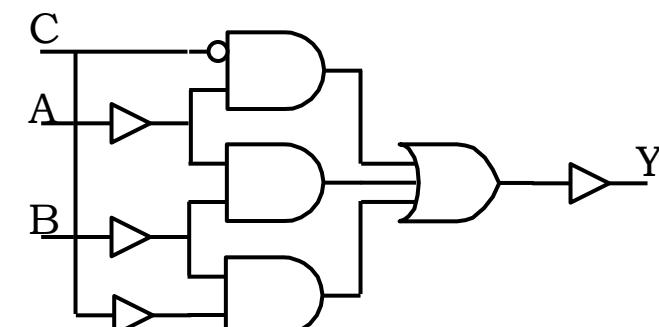


NOR-NOR



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

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



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

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



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: $\bar{a} + \bar{b} = \overline{ab}, \bar{a}\bar{b} = \overline{a + b}$

Boolean Minimization

Let's (again!) simplify

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

Using the identity

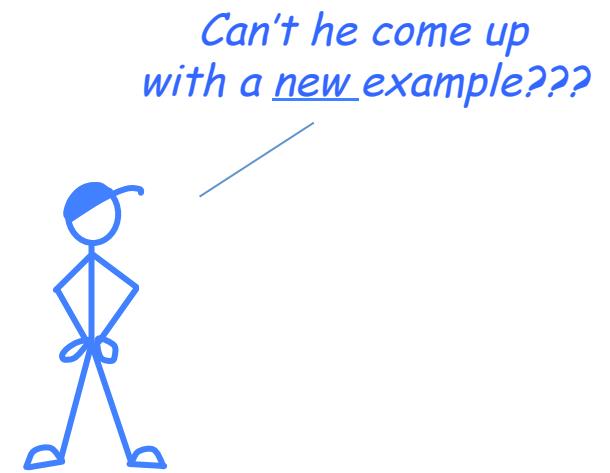
$$\alpha A + \alpha \overline{A} = \alpha(A + \overline{A}) = \alpha \cdot 1 = \alpha$$

For any expression α and variable A:

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

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

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



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

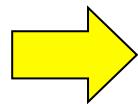


*Hey... I could write
a program to do
that*

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
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



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

$\rightarrow \bar{C}A$

$\rightarrow CB$

$\rightarrow BA$

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!

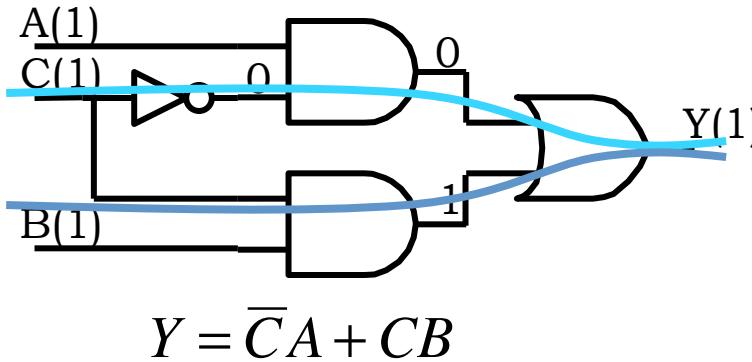
The Case for a Non-minimal SOP

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

\overline{CA}

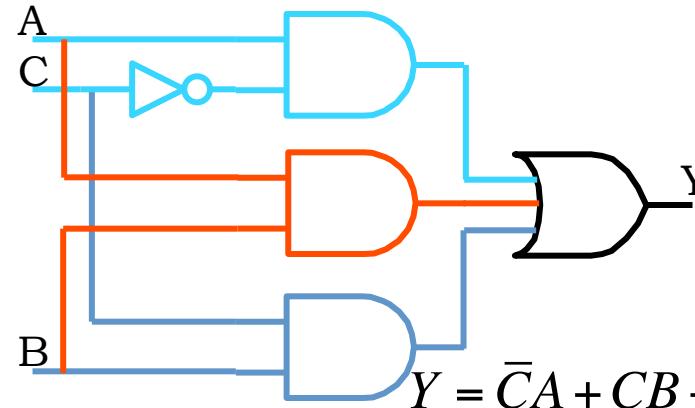
CB

BA



$$Y = \overline{CA} + CB$$

NOTE: The steady state behavior of these circuits is identical. They differ in their transient behavior.



$$Y = \overline{CA} + CB + AB$$



That's what we call a "glitch" or "hazard"



Now it's LENIENT!

Karnaugh Maps: A Geometric Approach

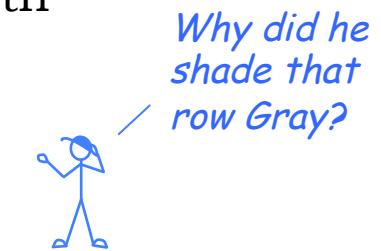
K-Map: a truth table arranged so that terms which differ by exactly one variable are adjacent to one another so we can see potential reductions easily.

Truth Table

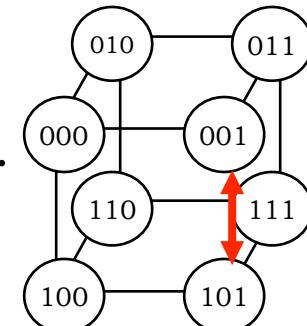
C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

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	1	0	1	0



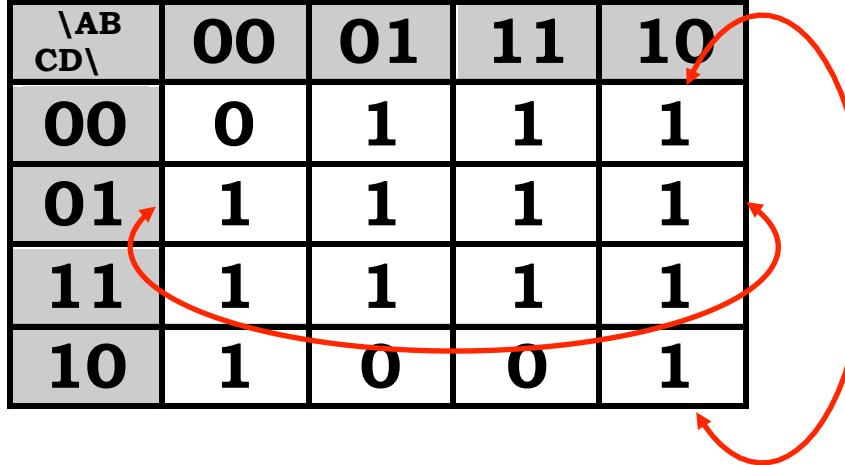
It's cyclic. The left edge is adjacent to the right edge.
(It's really just a flattened out cube).



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...

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

\overline{ABC}

$A\bar{C}$

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

\overline{AC}

\overline{B}

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

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

The Karnaugh map shows four prime implicants highlighted by red circles:

- A 2x2 square centered at (01, 11) containing cells (01, 11), (11, 11), (11, 10), and (01, 10).
- A 2x2 square centered at (00, 01) containing cells (00, 01), (01, 01), (01, 00), and (00, 00).
- A vertical column of length 2 starting at (11, 11) containing cells (11, 11) and (11, 10).
- A horizontal row of length 2 starting at (10, 11) containing cells (10, 11) and (10, 10).

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.

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

$$Y = A\bar{C} + BC$$

We're done!



\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

$$Y = D + B\bar{C} + A\bar{C} + \bar{B}\bar{C}$$

Minimal SOP is not necessarily unique!

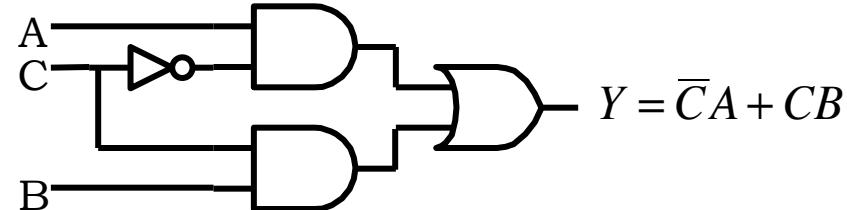


\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

$$Y = D + B\bar{C} + A\bar{B} + \bar{B}\bar{C}$$

Prime Implicants, Glitches & Leniency

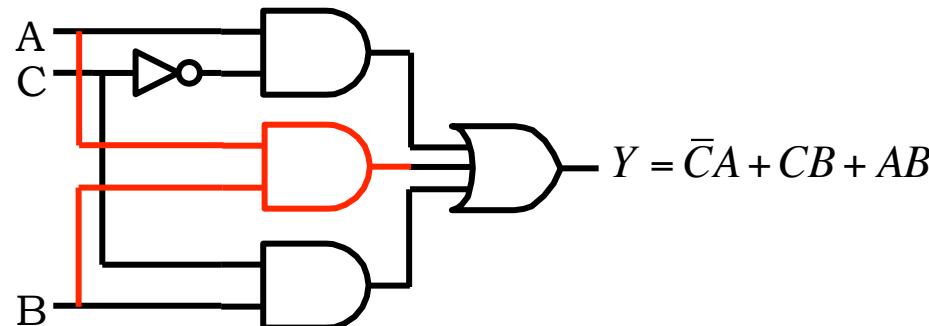
This circuit produces a glitch on Y when A=1, B=1, C: 1→0



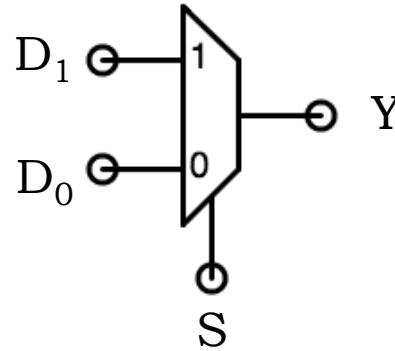
C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0



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



We've Been Designing a Mux

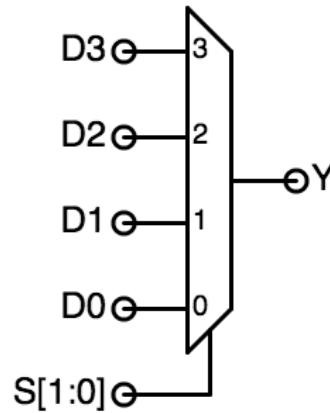


2-input Multiplexer

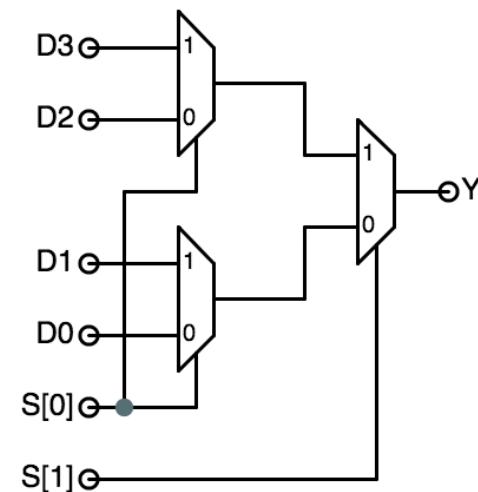
Truth Table

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 ...

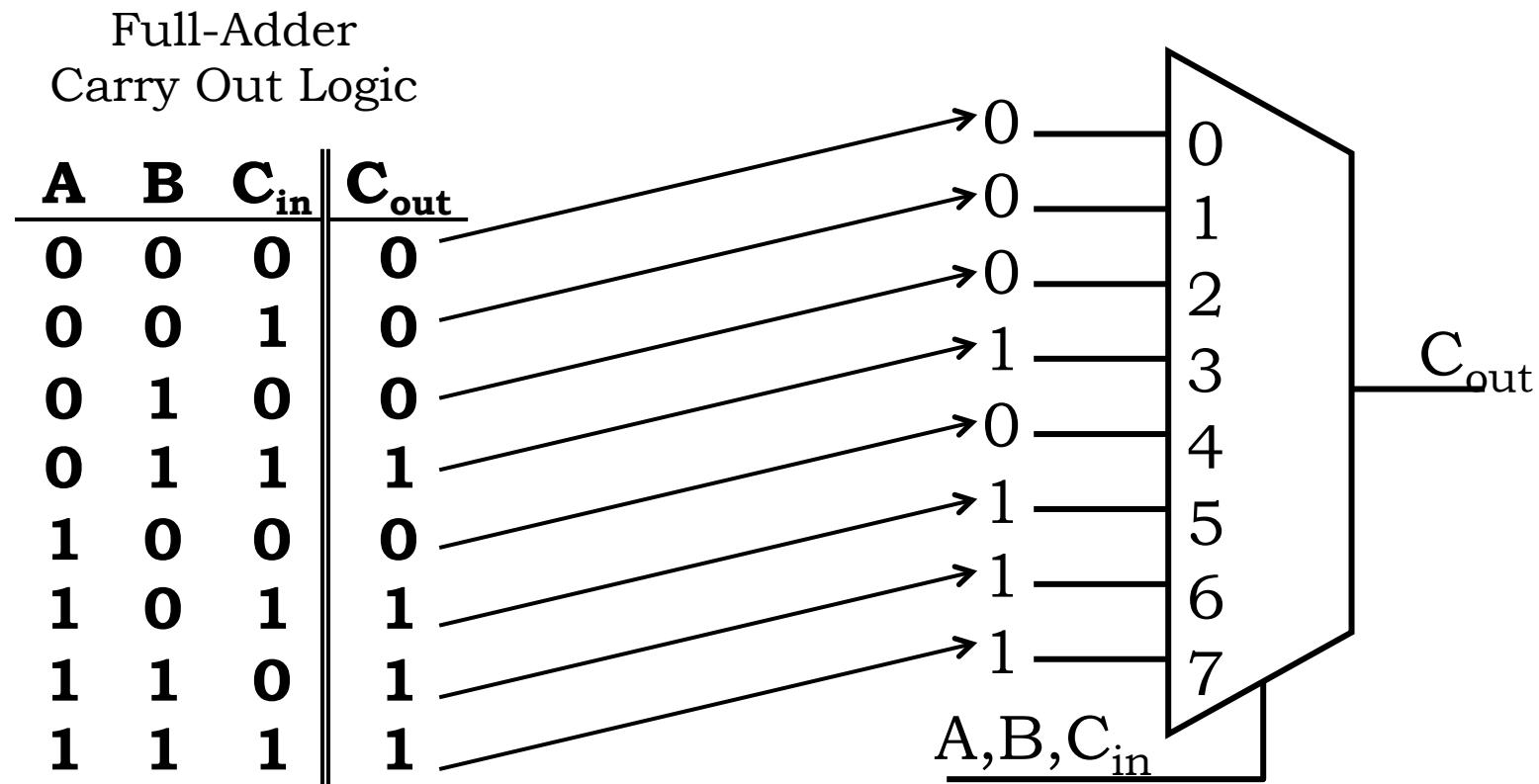


... and implemented as a tree of smaller MUXes:

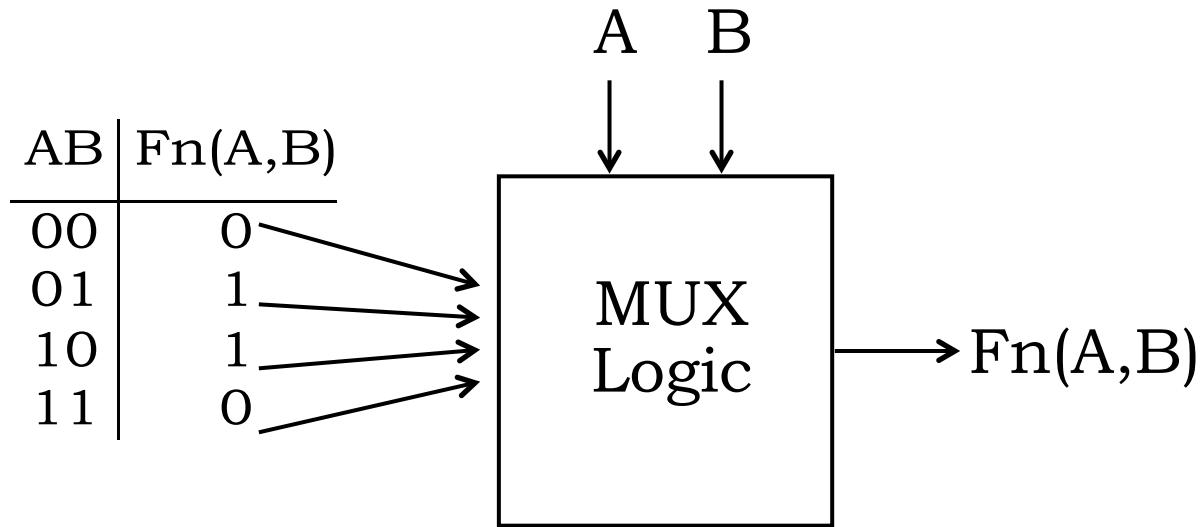


Systematic Implementation Strategies

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



Synthesis By Table Lookup



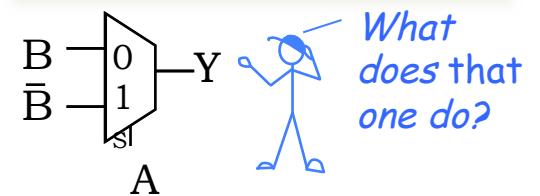
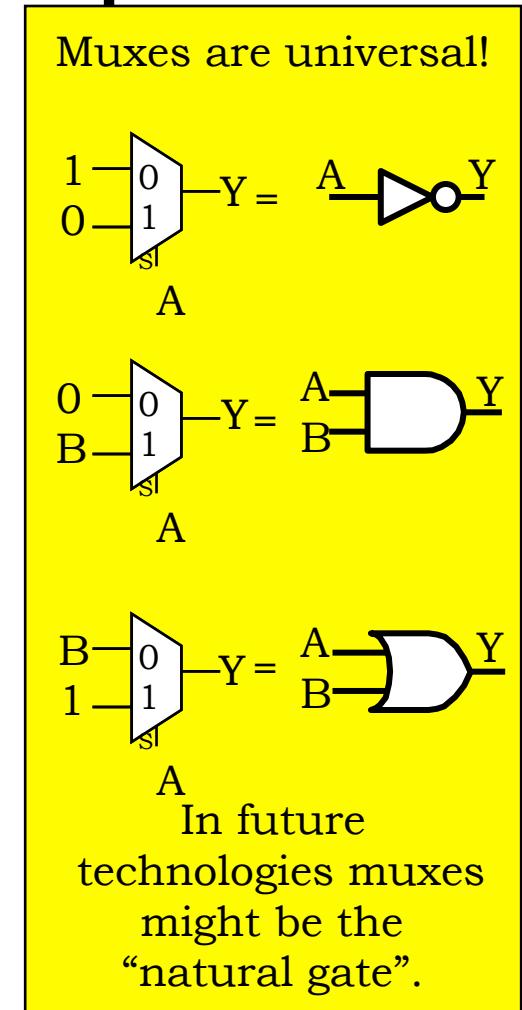
Generalizing:

In theory, we can build any 1-output combinational logic block with multiplexers.

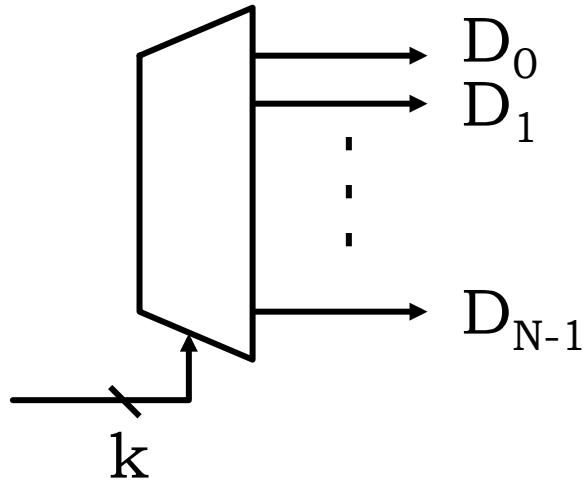
For an N -input function we need a 2^N input mux.

Is this practical for BIG truth tables?

How about 10-input function? 20-input?



A New Combinational Device

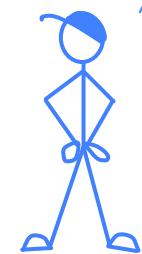


DECODER:

- 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'

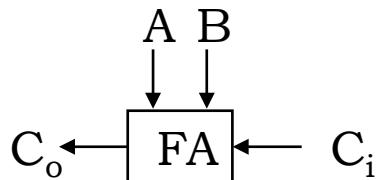


NOW, we are well on our way to building a general purpose table-lookup device.

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

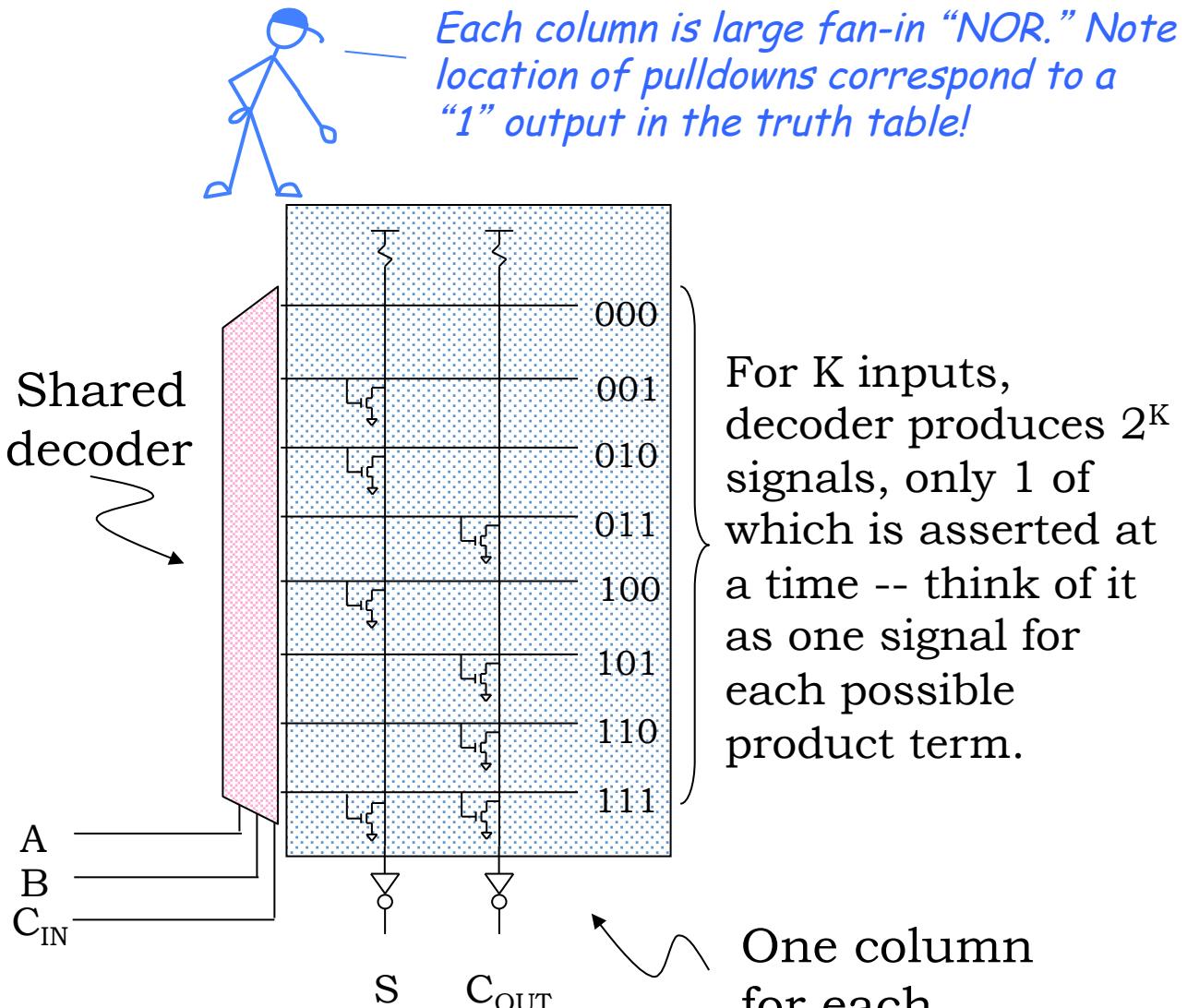
Read-only Memory (ROM)

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

Shared
decoder



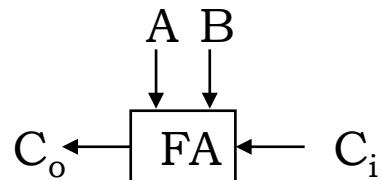
Each column is large fan-in “NOR.” Note location of pulldowns correspond to a “1” output in the truth table!

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

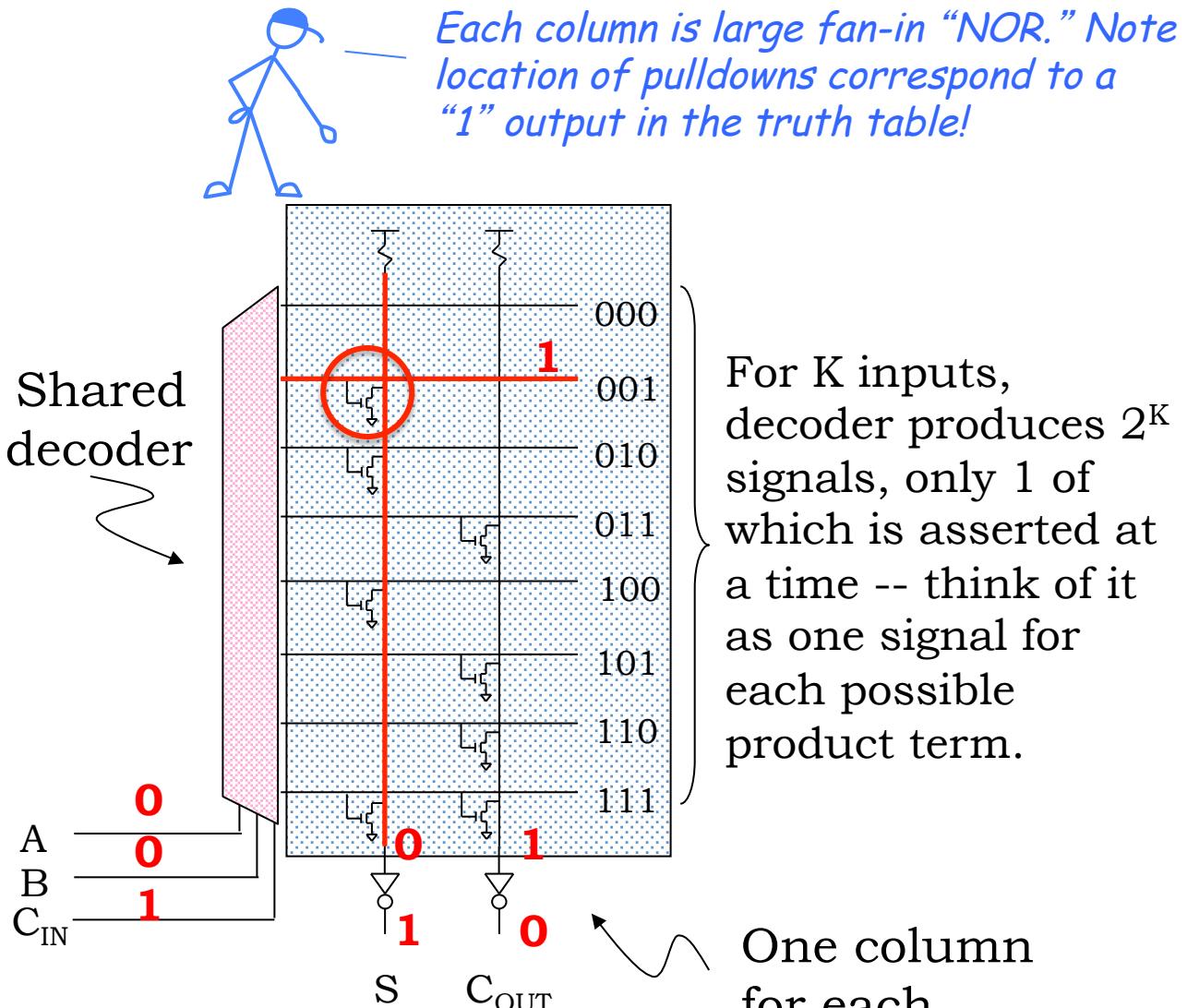
Read-only Memory (ROM)

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

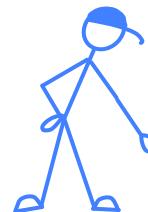
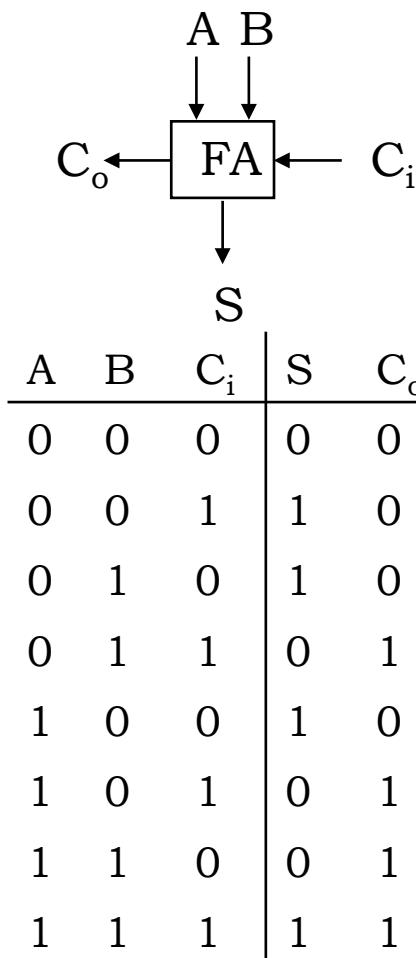
Shared decoder



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

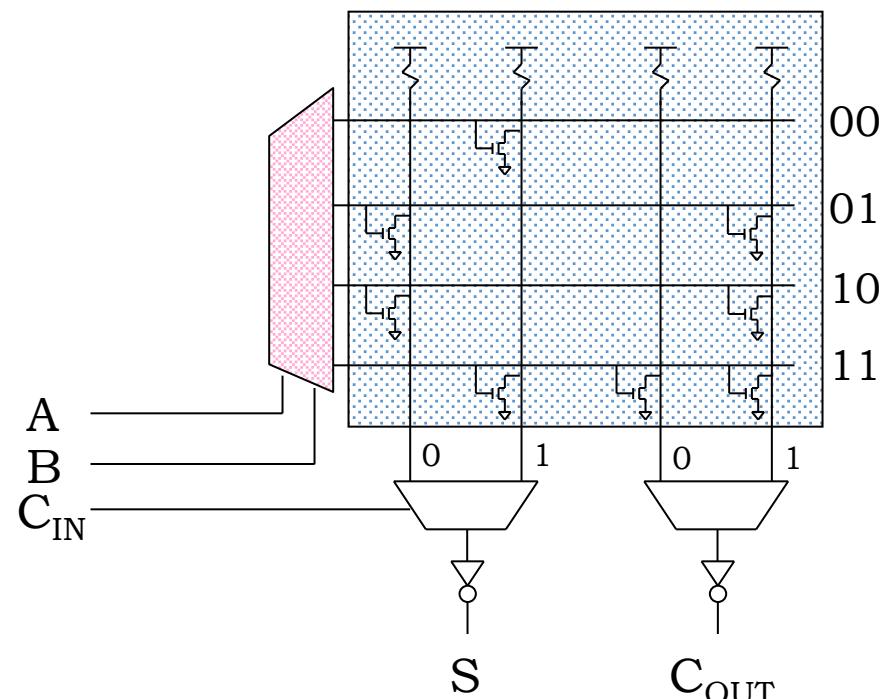
Read-only Memory (ROM)

Full Adder



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...

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 $\approx \underline{2^N \times \#outputs}$

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

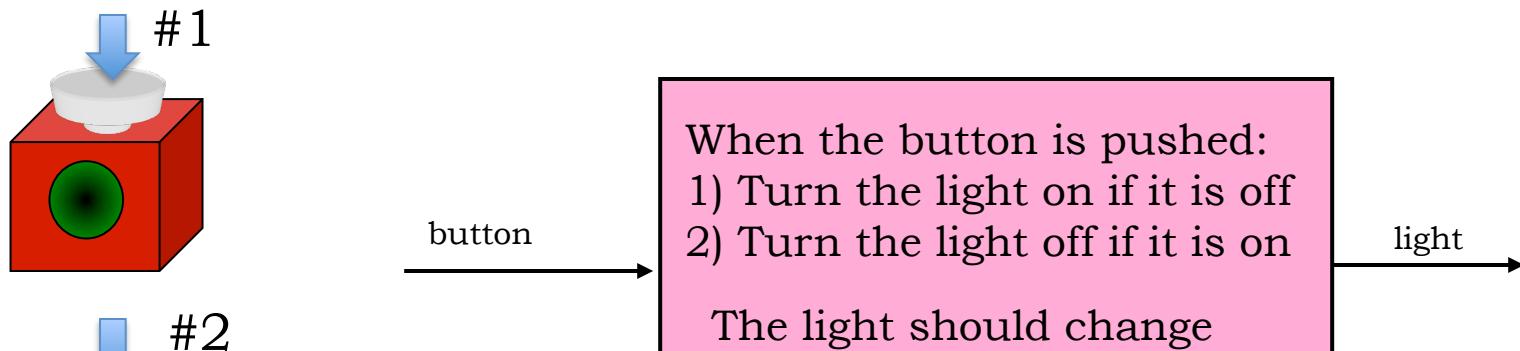
5. Sequential Logic

6.004x Computation Structures
Part 1 – Digital Circuits

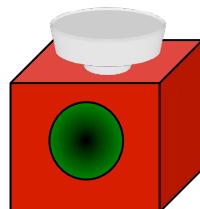
Copyright © 2015 MIT EECS

Something We Can't Build (Yet)

What if you were given the following design specification:

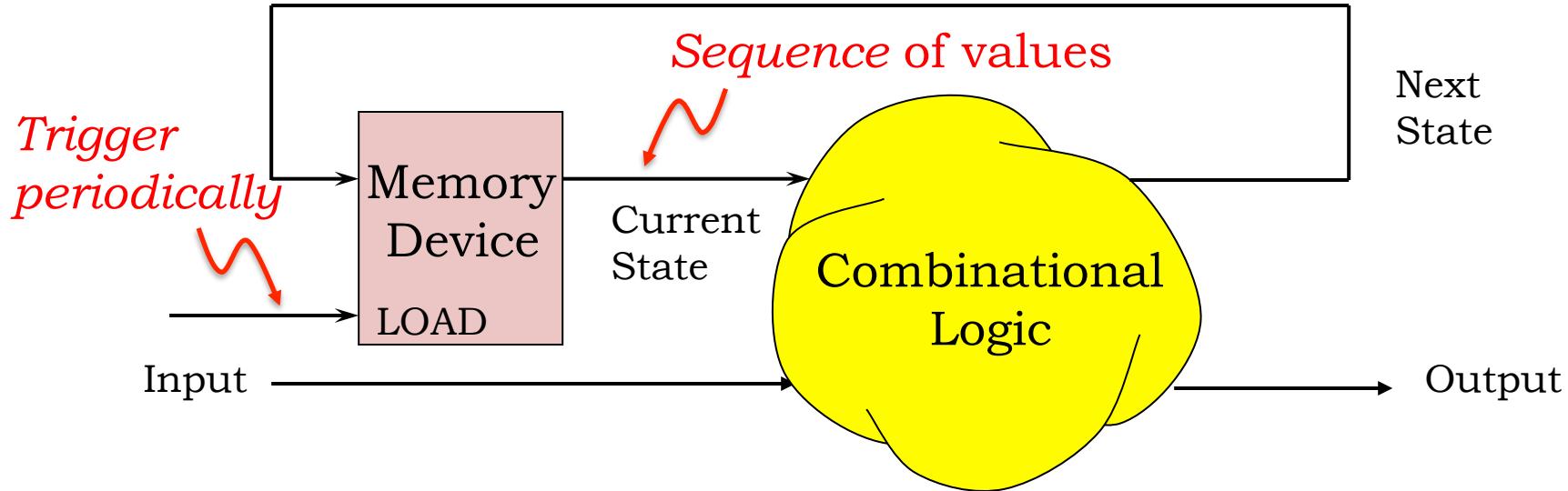


What makes this device different from those we've discussed before?



1. “State” – i.e., the device has memory
2. The output was changed by a input “event” (pushing a button) rather than an input “level”

Digital State: What We'd Like to Build



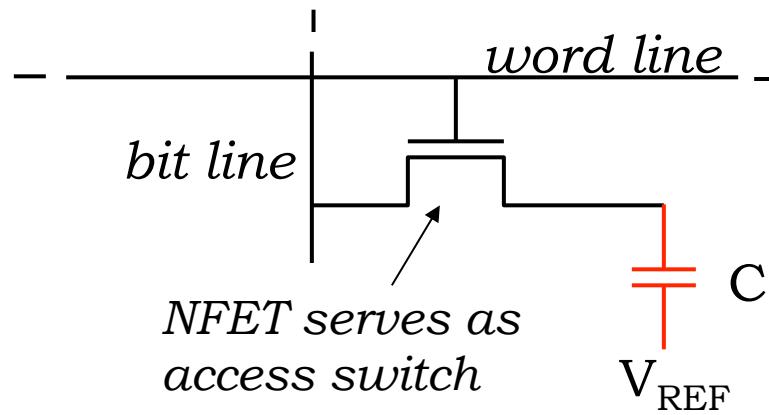
Plan: Build a Sequential Circuit with stored digital STATE –

- 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



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:

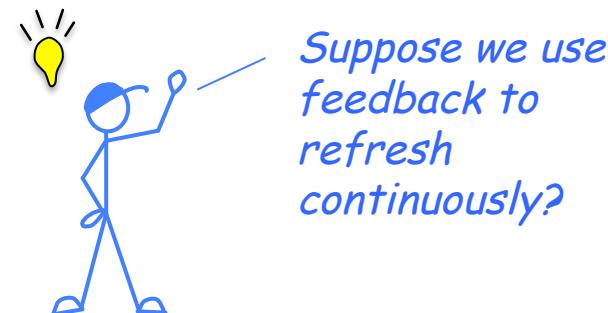
precharge bit line, turn on access fet,
detect (small) change in bit line voltage

Pros:

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

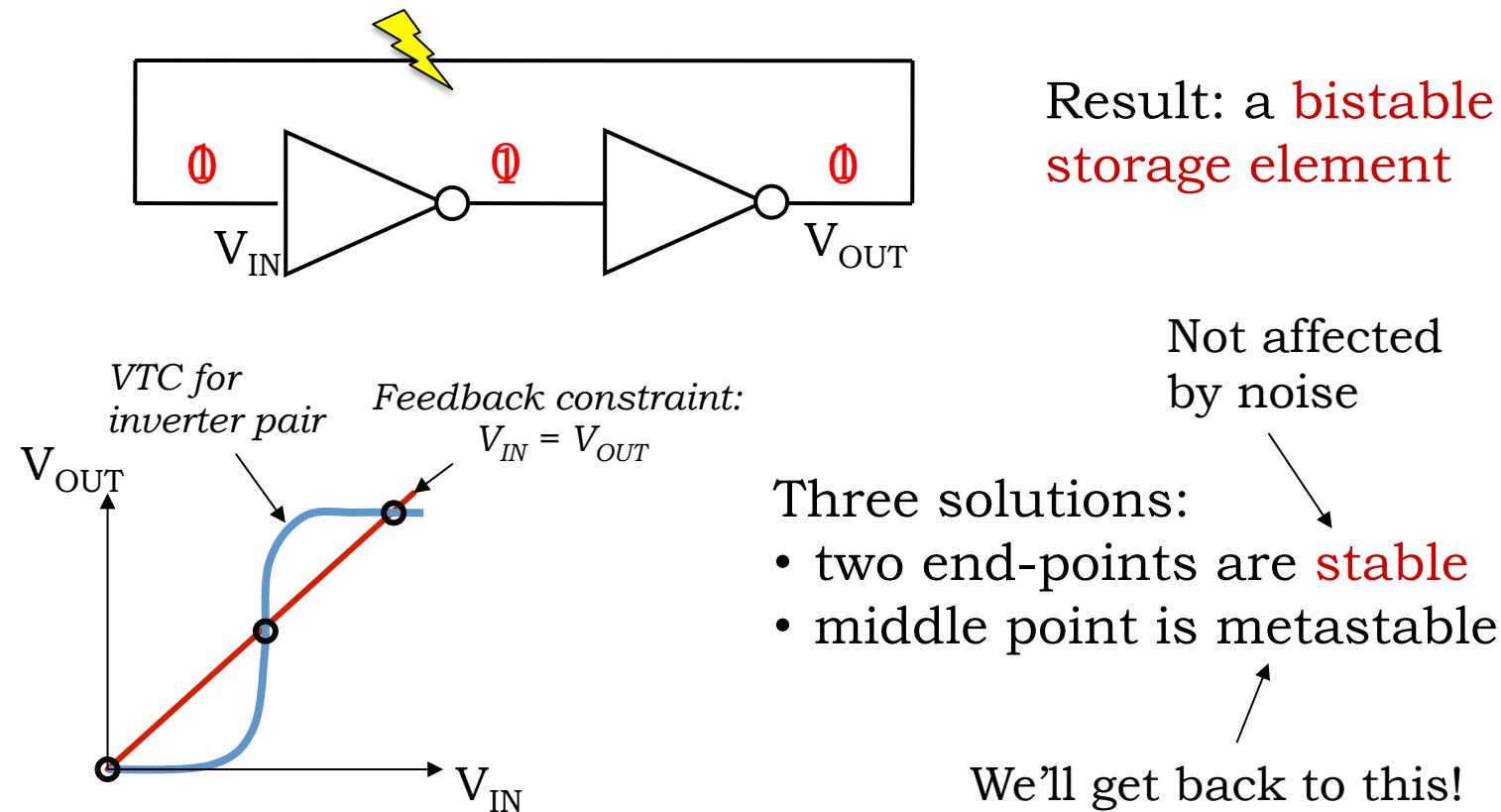
Cons:

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



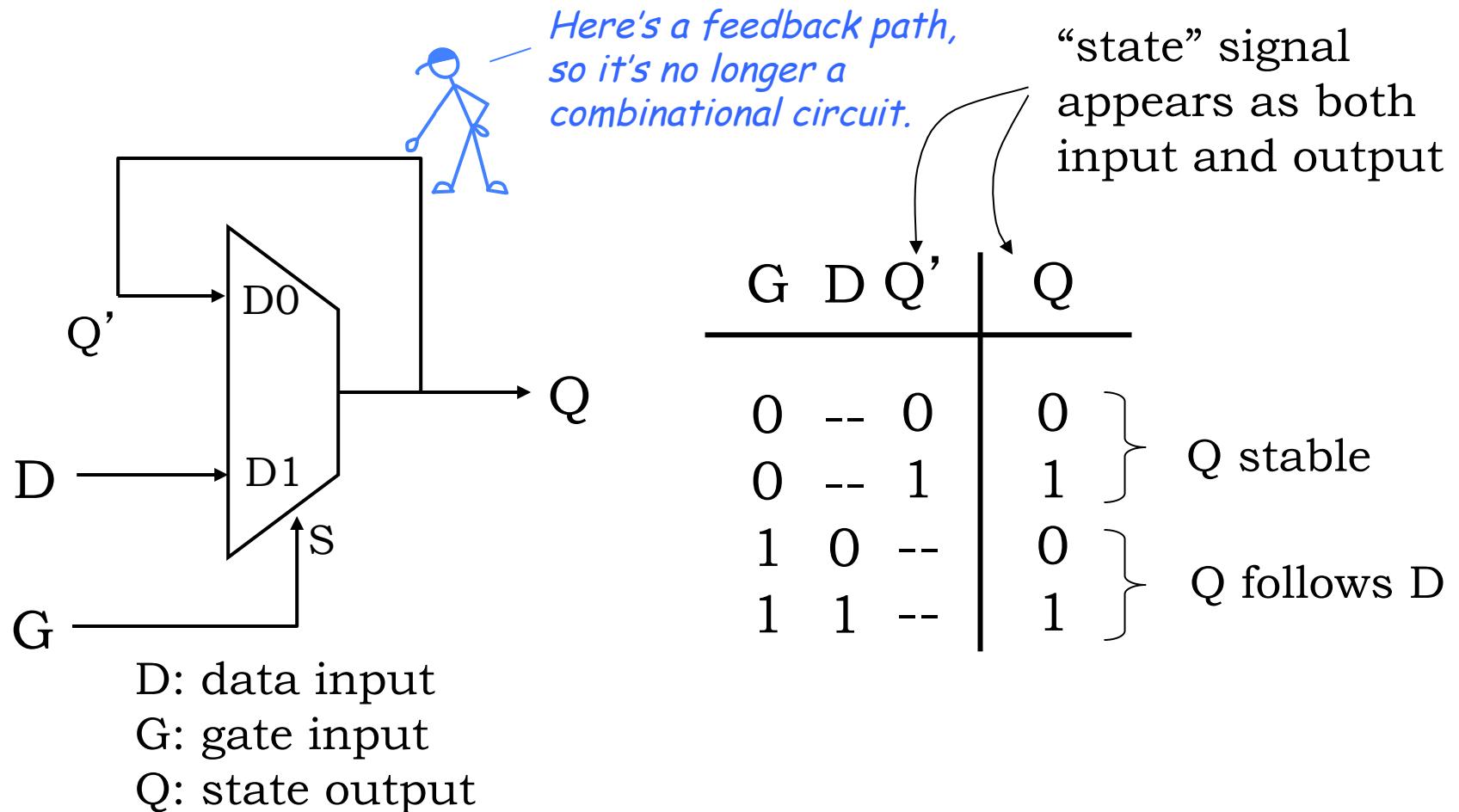
Memory: Using Feedback

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!

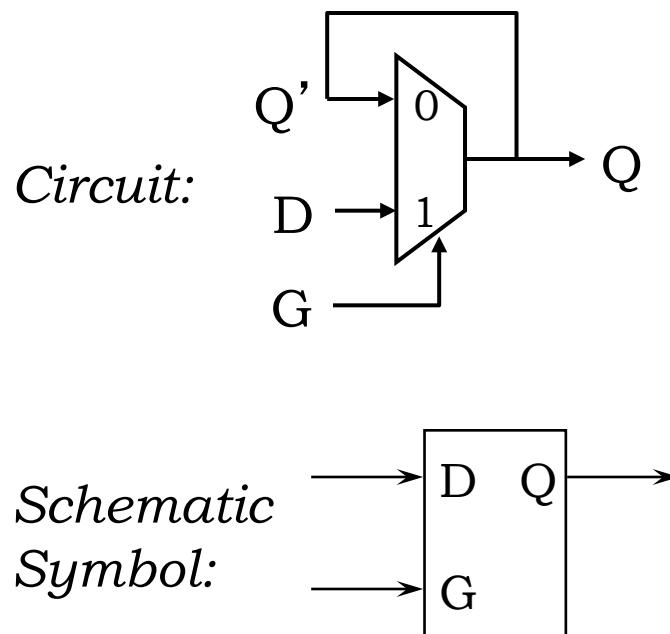


Settable Memory Element

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



New Device: D Latch

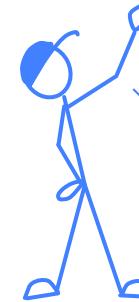
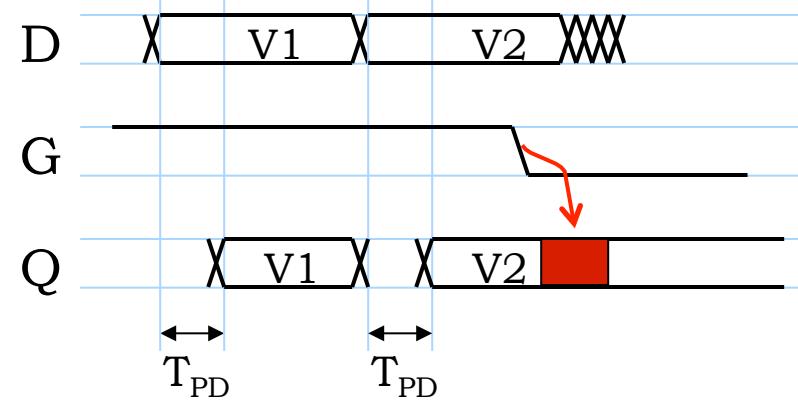


$G=1$:

Q follows D

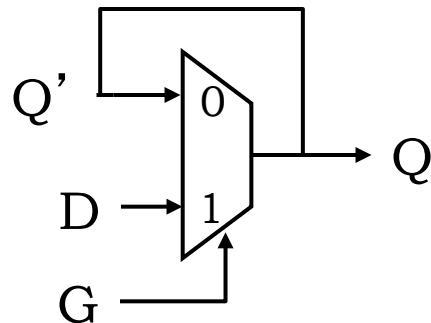
$G=0$:

Q holds

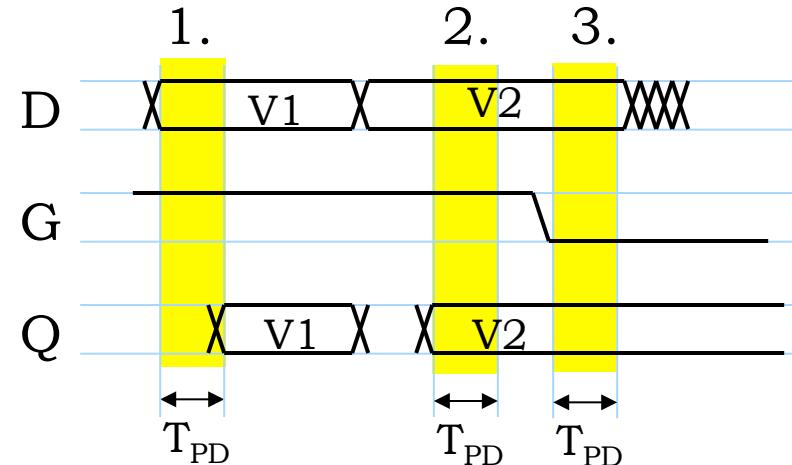


BUT... A change in D or G contaminates Q , hence Q' ... how can this possibly work?

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

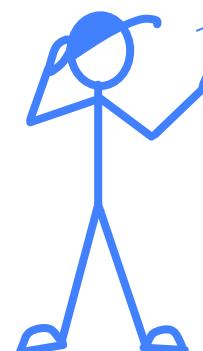


Assume LENIENT Mux,
propagation delay of T_{PD}

Then output valid when

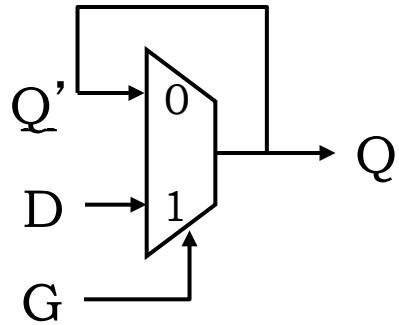
1. $G=1$, D stable for T_{PD} , *independently of Q'* ; or
2. $Q=D$ stable for T_{PD} , *independently of G*; or
3. $G=0$, Q stable for T_{PD} , *independently of D*

Does lenience *guarantee* a working latch?



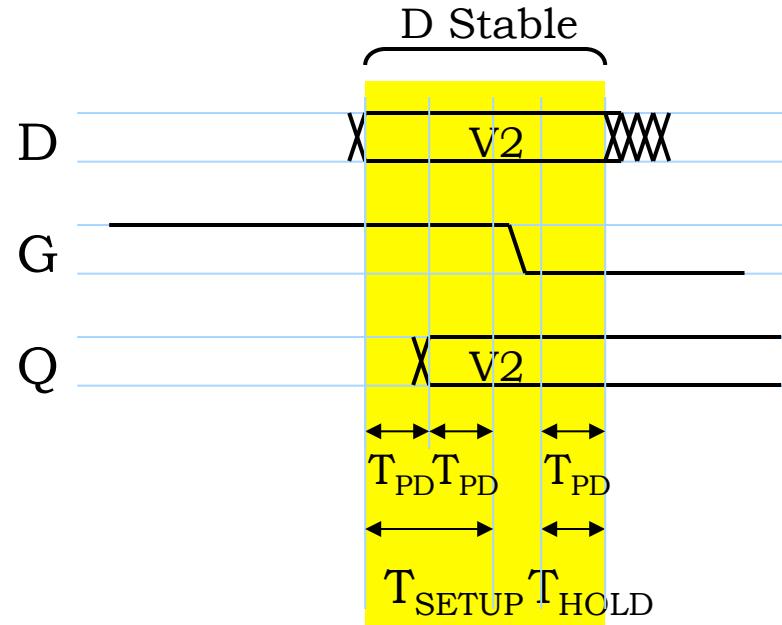
What if D and G change at about the same time...

...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*

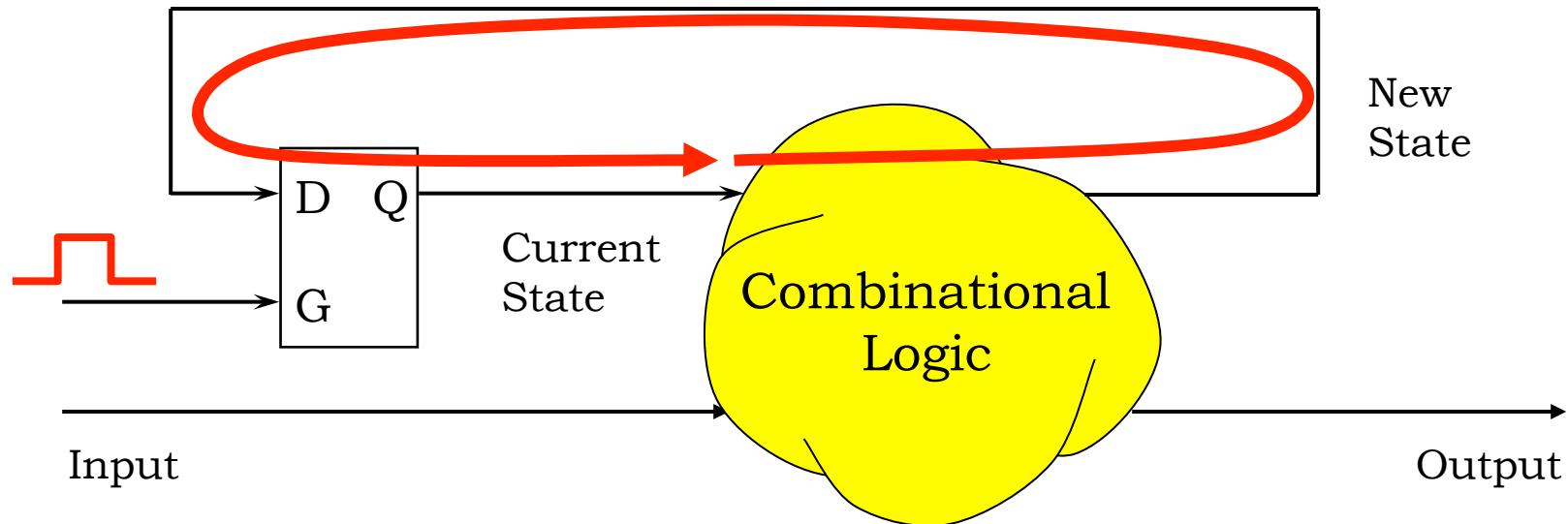


Dynamic Discipline for our latch:

$T_{SETUP} = 2T_{PD}$: 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

Let's Try It Out!



When $G=1$, latch is *Transparent*...

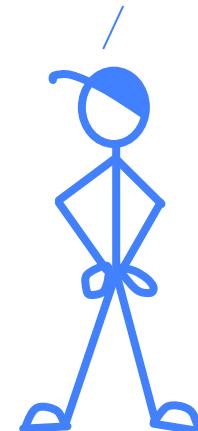
... provides a combinational path from D to Q.

Can't work without tricky timing constraints on $G=1$ pulse:

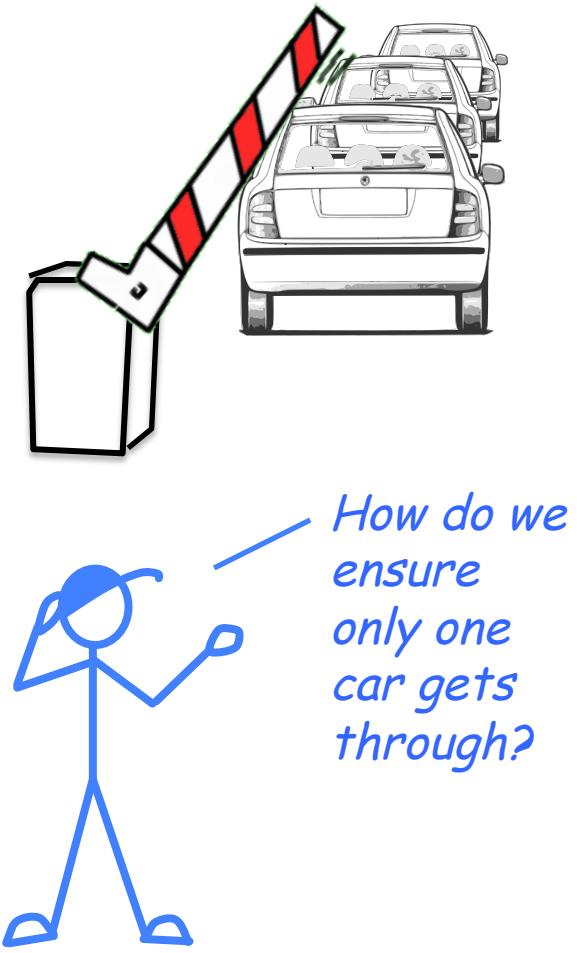
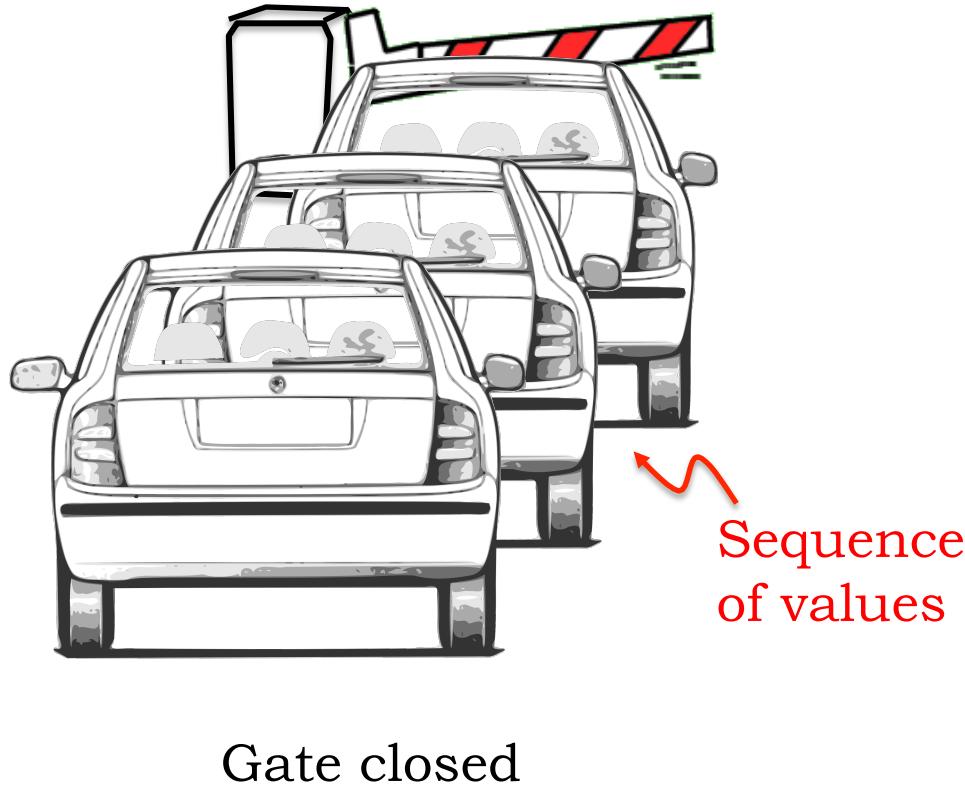
- Must fit within contamination delay of logic
- Must accommodate latch setup, hold times

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

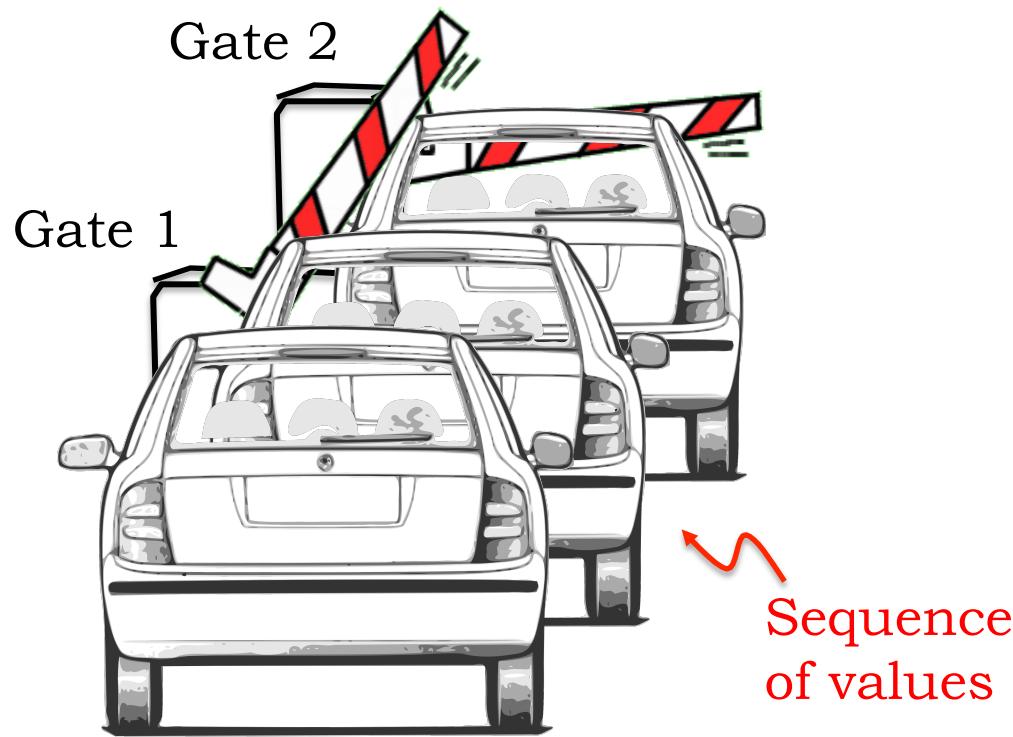
Looks like a stupid approach to me...



Flakey Control Systems



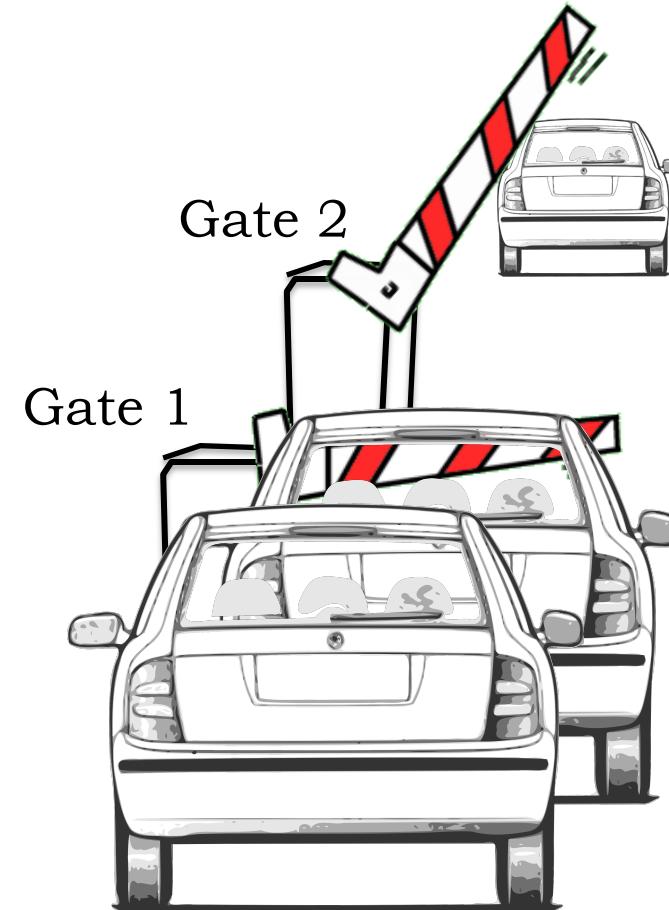
Solution: Escapement Strategy (2 gates)



Gate 1: open
Gate 2: closed

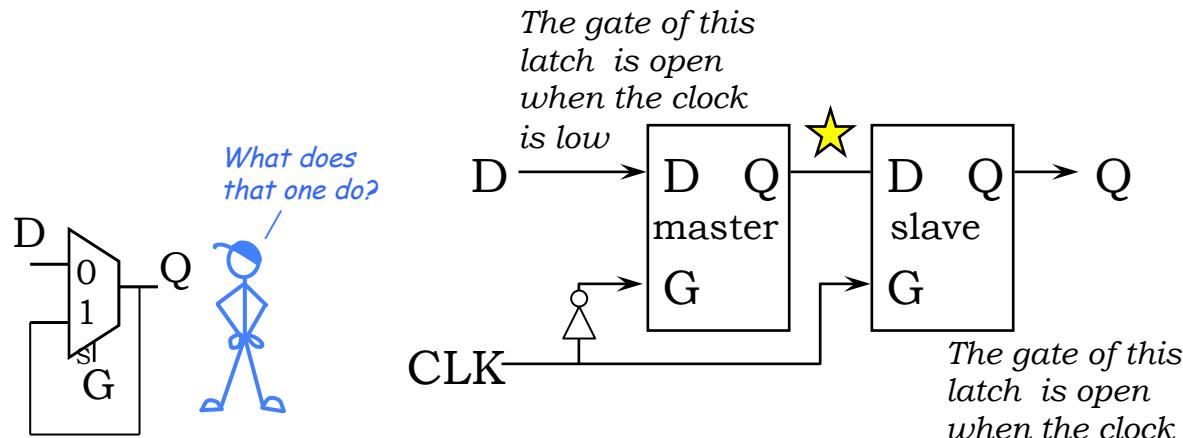


Key: at no time is there a path through both gates



Gate 1: closed
Gate 2: open

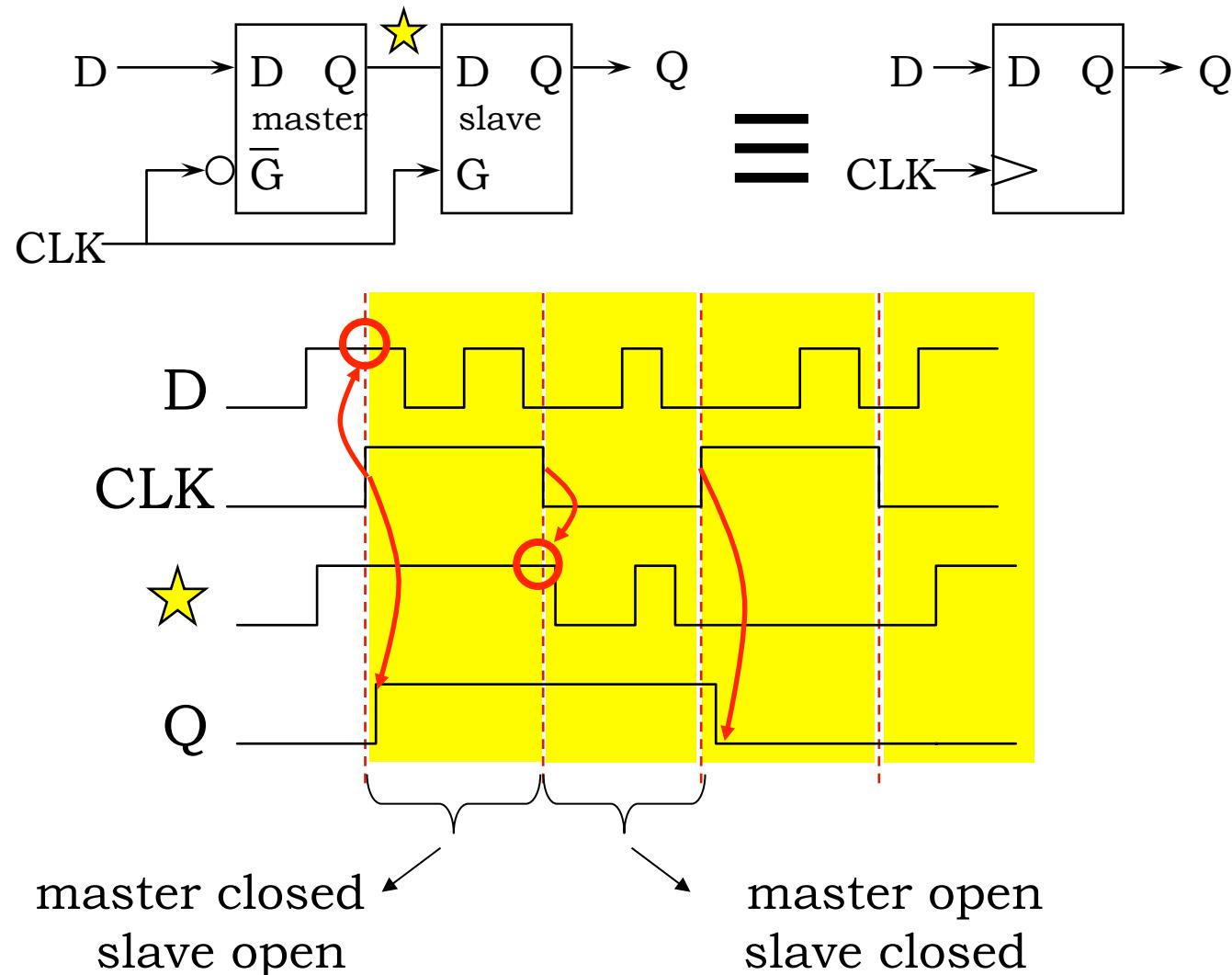
(Edge-Triggered) D Register



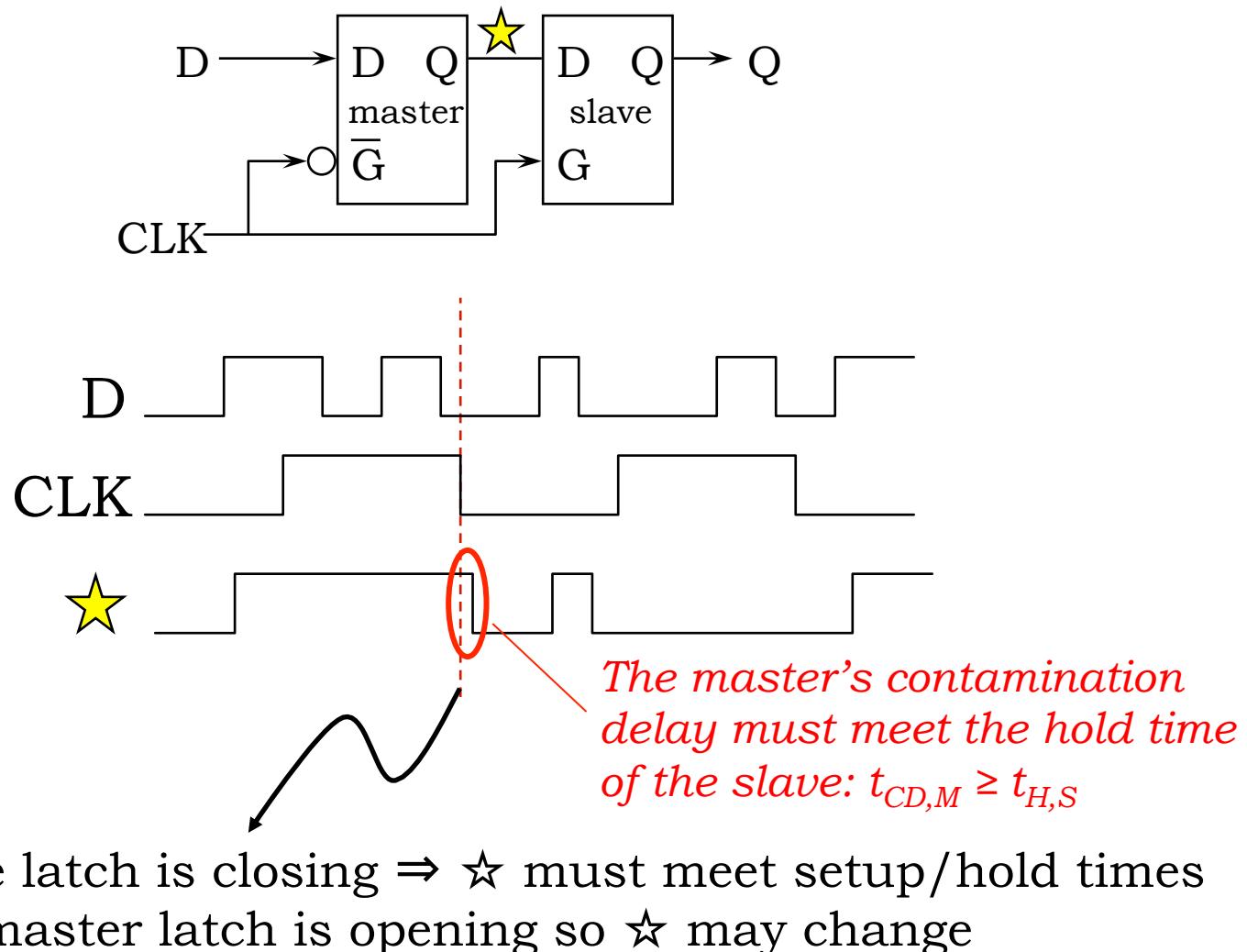
Observations:

- 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)

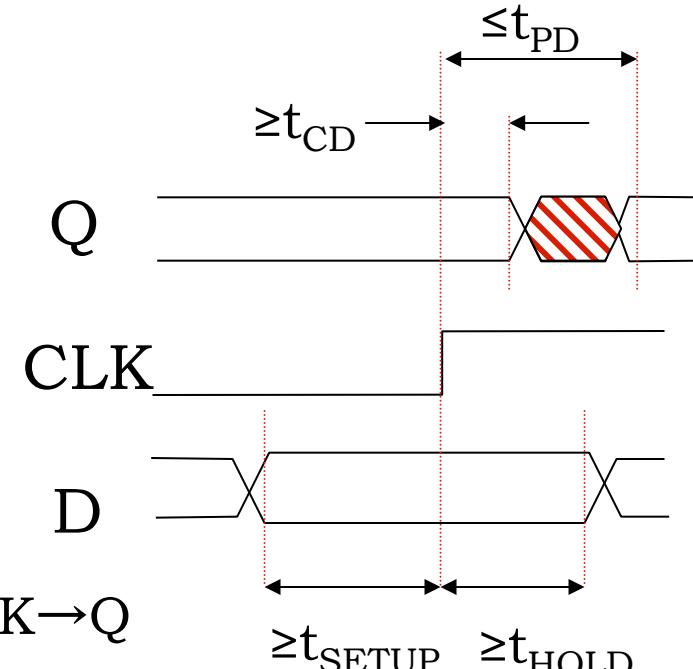
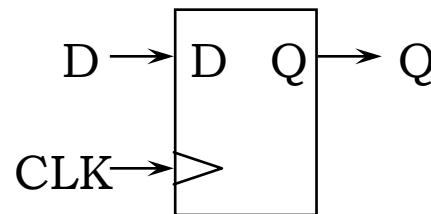
D-Register Waveforms



Um, about that hold time...



D-Register Timing 1



t_{PD} : maximum propagation delay, $CLK \rightarrow Q$

t_{CD} : minimum contamination delay, $CLK \rightarrow Q$

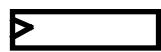
t_{SETUP} : setup time

guarantee that D has propagated through feedback path before master closes

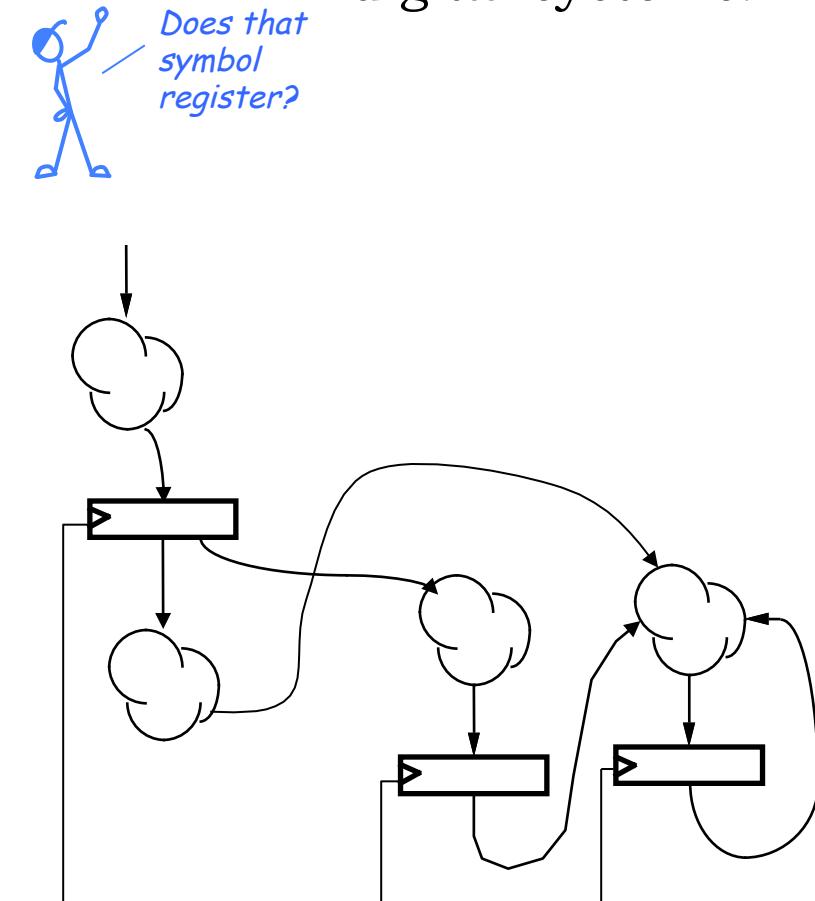
t_{HOLD} : hold time

guarantee master is closed and data is stable before allowing D to change

Single-clock Synchronous Circuits



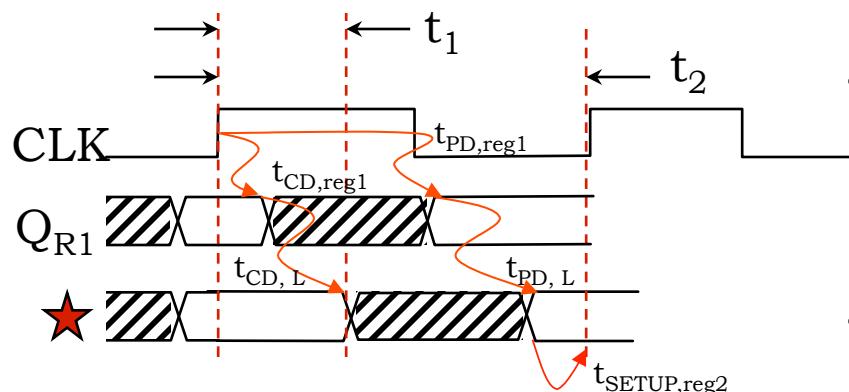
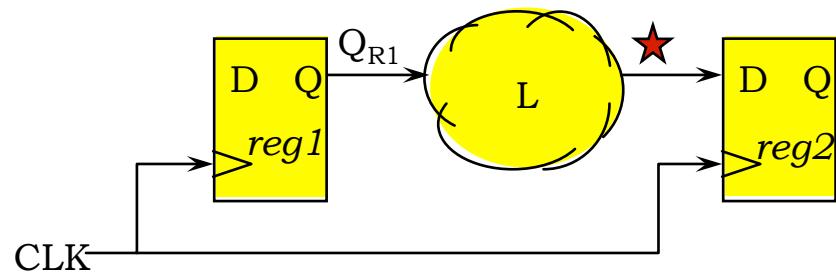
We'll use registers in a highly constrained way to build digital systems:



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!

Timing in a Single-clock System



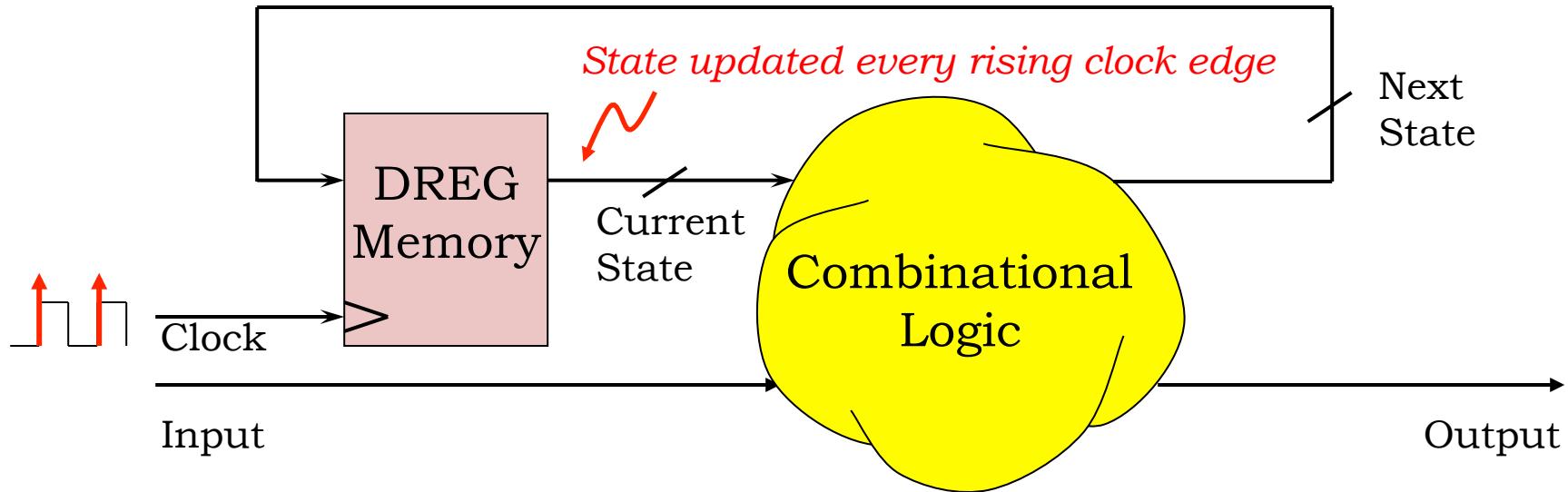
$$t_1 = t_{CD, reg1} + t_{CD, L} \geq t_{HOLD, reg2}$$

$$t_2 = t_{PD, reg1} + t_{PD, L} + t_{SETUP, reg2} \leq t_{CLK}$$

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")?

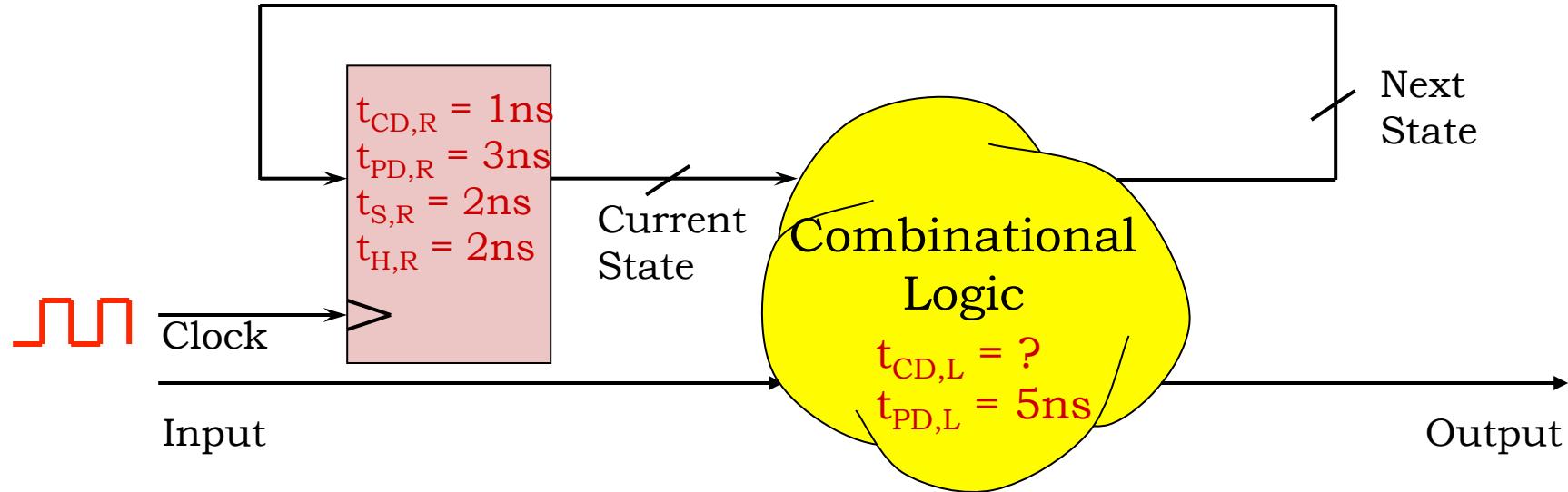
Model: Discrete Time



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!)

Sequential Circuit Timing



Questions:

- Constraints on t_{CD} for the logic?
- Minimum clock period?
- Setup, Hold times for Inputs?

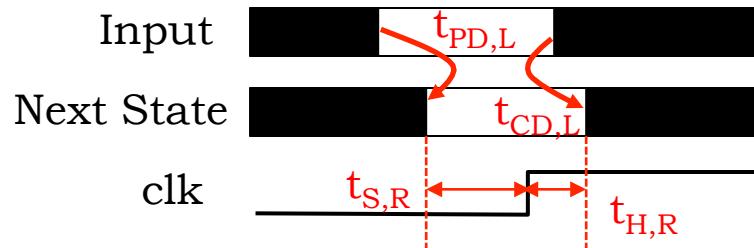
$$t_{S,INPUT} = t_{PD,L} + t_{S,R} = 7 \text{ nS}$$

$$t_{H,INPUT} = t_{H,R} - t_{CD,L} = 1 \text{ nS}$$

$$t_{CD,R} (1 \text{ ns}) + t_{CD,L}(?) \geq t_{H,R}(2 \text{ ns})$$

$$t_{CD,L} \geq 1 \text{ ns}$$

$$t_{CLK} \geq t_{PD,R} + t_{PD,L} + t_{S,R} = 10\text{nS}$$



Summary

Basic memory elements:

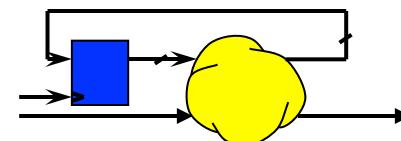
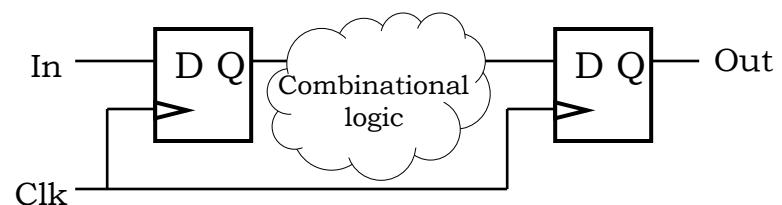
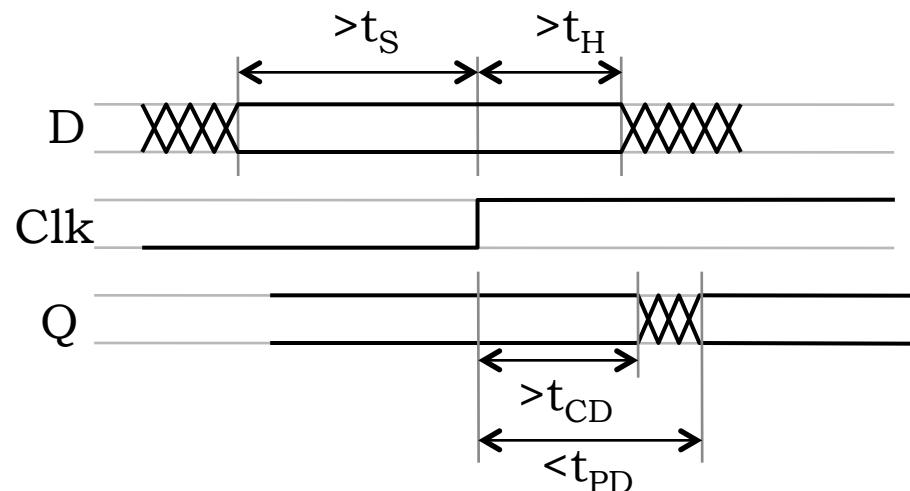
- Feedback, detailed analysis
=> basic level-sensitive devices (eg, latch)
- 2 Latches => Register
- Dynamic Discipline: constraints on input timing

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!

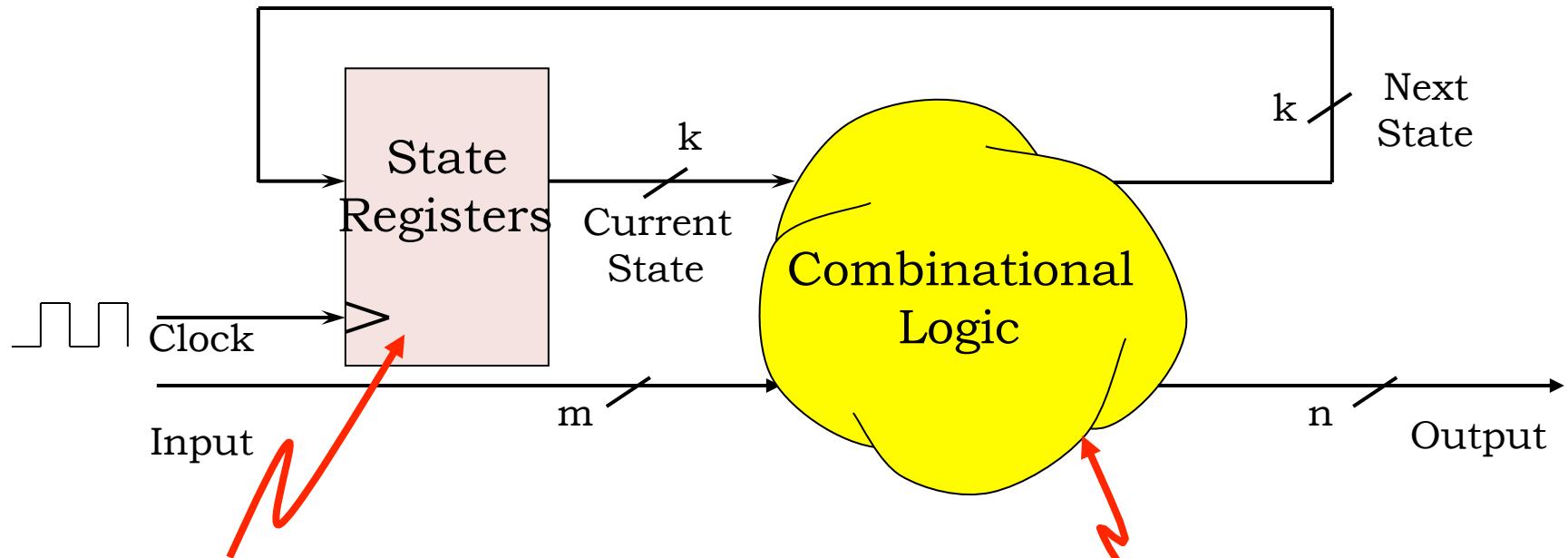


6. Finite State Machines

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS

Our New Machine

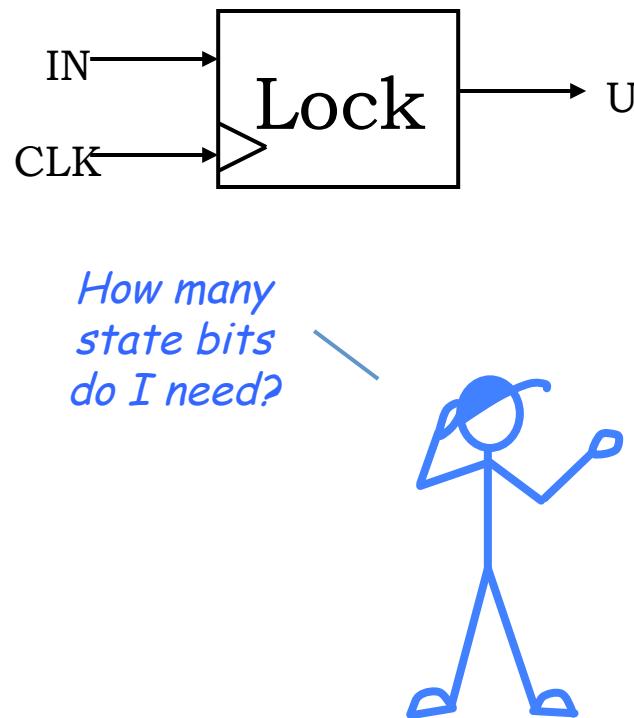


- Engineered cycles
- Works only if dynamic discipline obeyed
- Remembers k bits for a total of 2^k unique combinations
- Acyclic graph
- Obeys static discipline
- Can be exhaustively enumerated by a truth table of 2^{k+m} rows and $k+n$ output columns

A Simple Sequential Circuit...

Lets make a digital binary *Combination Lock*:

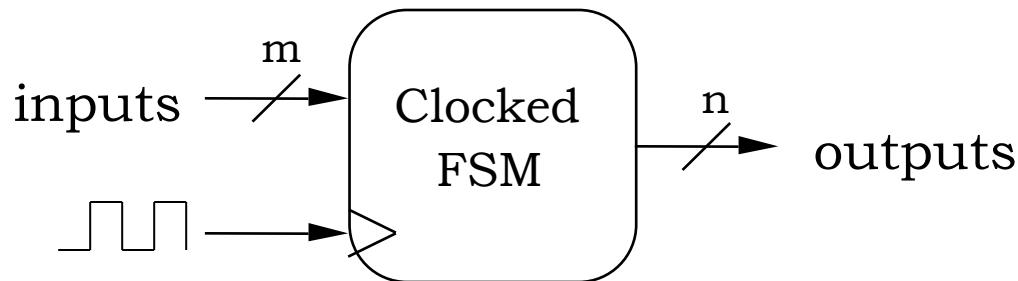
Specification:



- One input ("0" or "1")
- One output ("Unlock" signal)
- UNLOCK is 1 if and only if:

Last 4 inputs were the
“combination”: 0110

Abstraction du jour: Finite State Machines

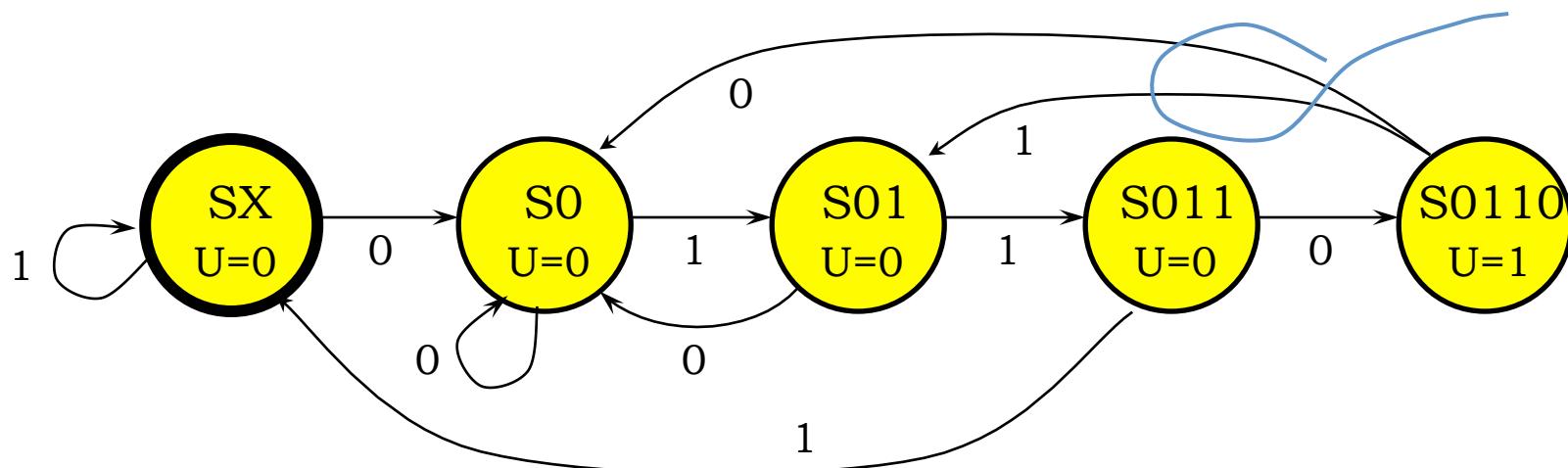


A FINITE STATE MACHINE has

- 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: $\text{Out}(s)$ or $\text{Out}(s, I)$ for each state s and input I

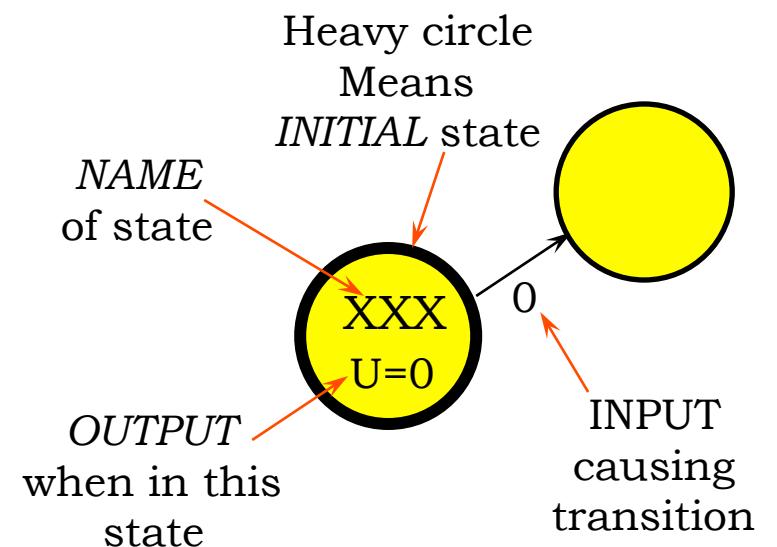
State Transition Diagram

Why do these
go to S0 and S01?

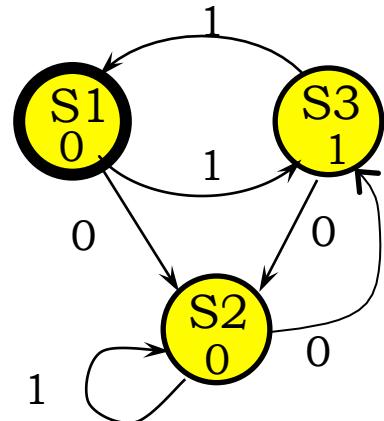


Designing our lock ...

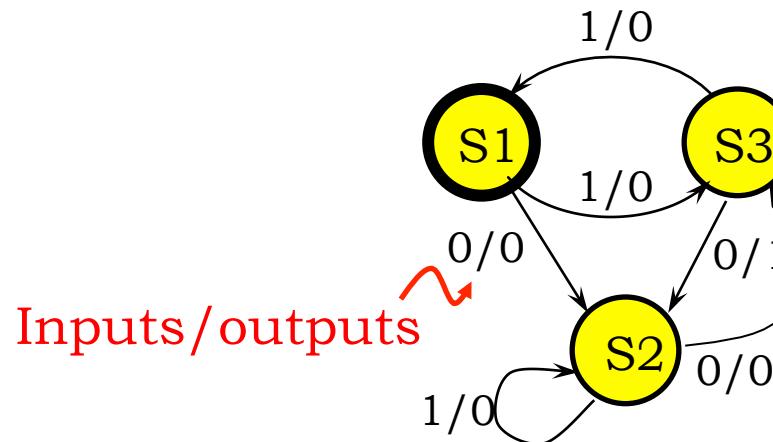
- Need an initial state; call it SX.
- Must have a separate state for each step of the proper entry sequence
- Must handle other (erroneous) entries



Valid State Diagrams



MOORE Machine:
Outputs on States



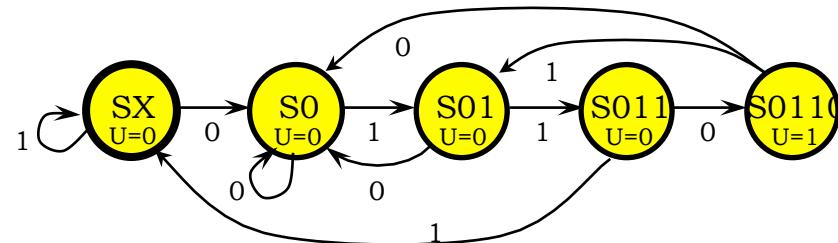
Inputs/outputs

MEALY Machine:
Outputs on Transitions

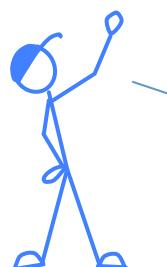
Arcs leaving a state must be:

- (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.*

State Transition Diagram as a Truth Table

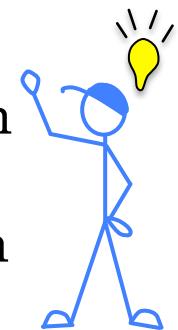


IN	Current State		Next State	Unlock	
0	000	SX	S0	001	0
1	000	SX	SX	000	0
0	001	S0	S0	001	0
1	001	S0	S01	011	0
0	011	S01	S0	001	0
1	011	S01	S011	010	0
0	010	S011	S0110	100	0
1	010	S011	SX	000	0
0	100	S0110	S0	001	1
1	100	S0110	S01	011	1



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...

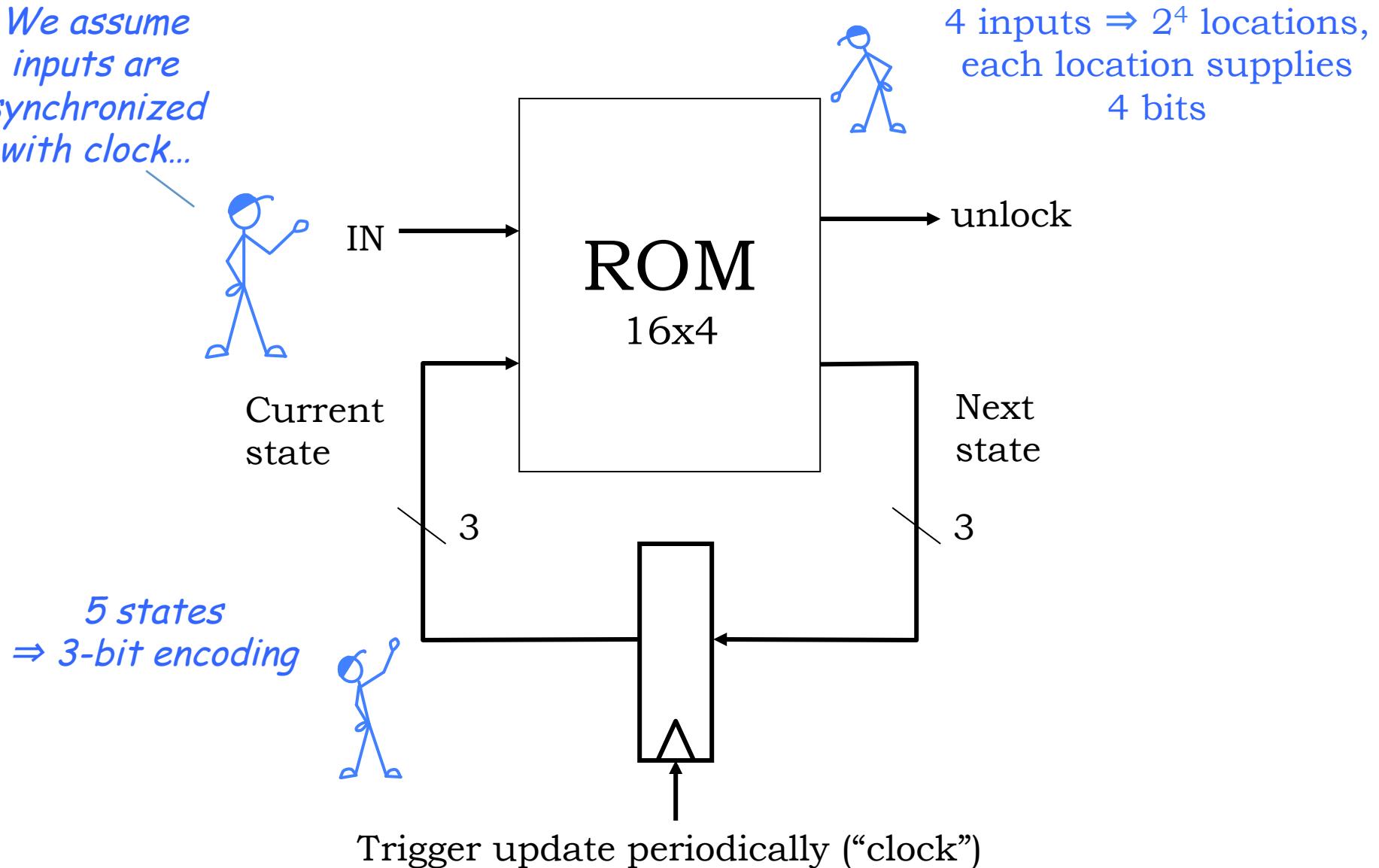


Binary encodings are assigned to each state (a bit of art)

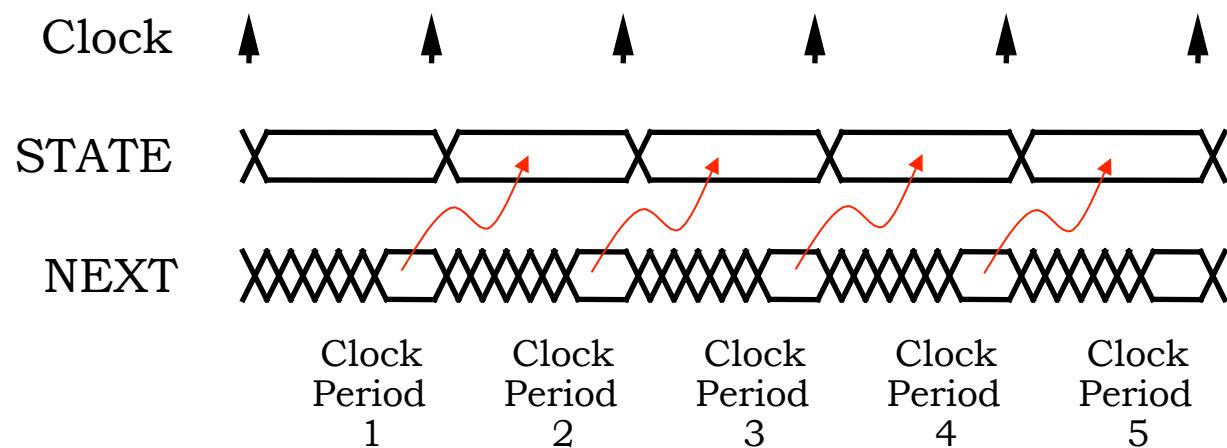
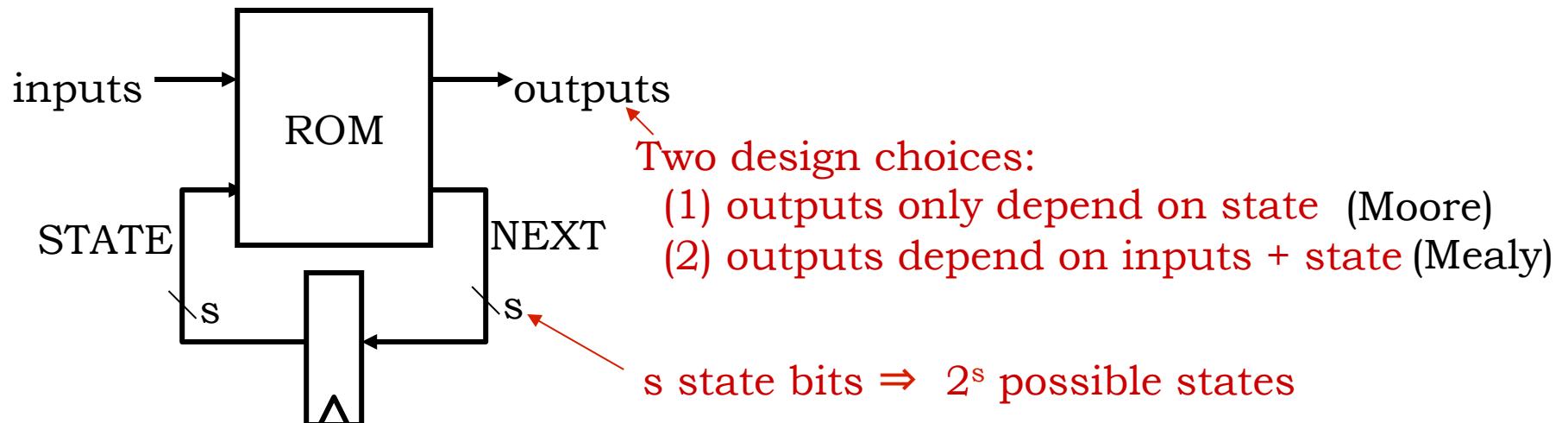
The truth table can then be simplified using the reduction techniques we learned for combinational logic

Now Put It In Hardware!

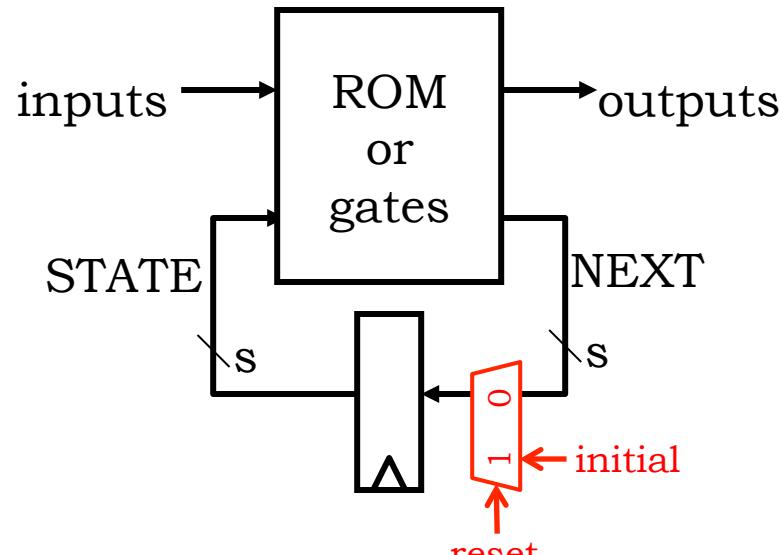
We assume
inputs are
synchronized
with clock...



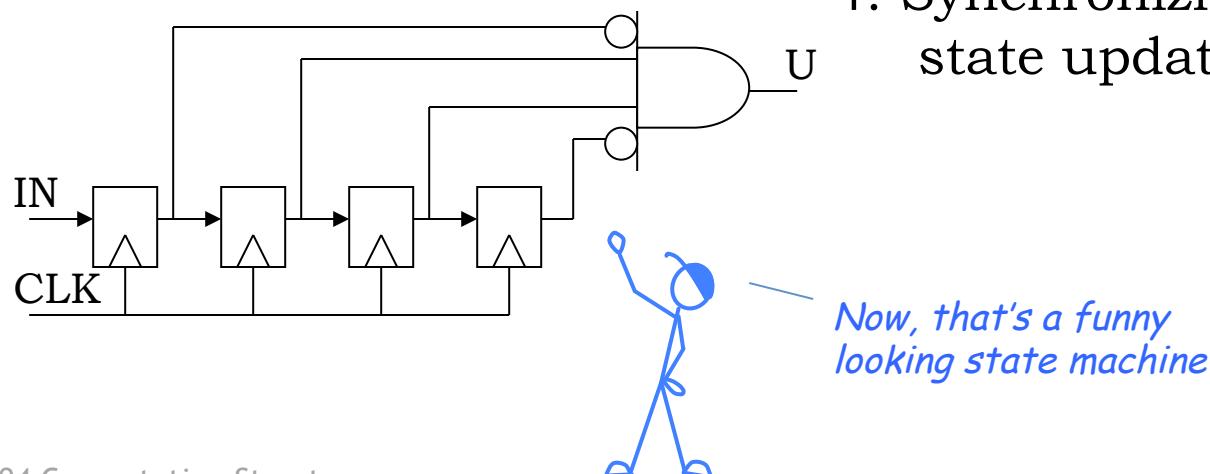
Discrete State, Discrete Time



Housekeeping Issues...

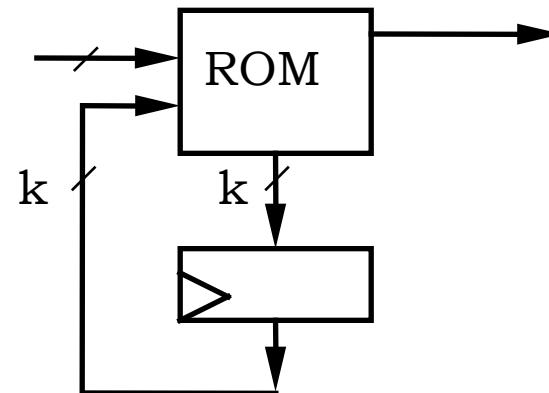


1. Initialization? Clear the memory?
2. Unused state encodings?
 - waste ROM (use gates)
 - what does it mean?
 - can the FSM recover?
3. Choosing encoding for state?
4. Synchronizing input changes with state update?

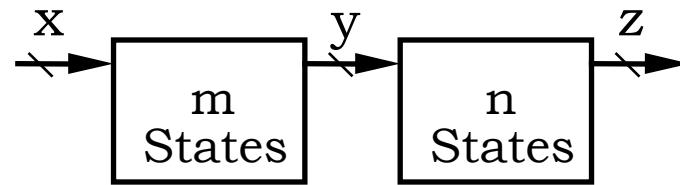


FSM States

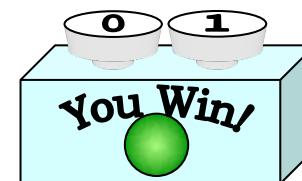
1. What can you say about the number of states?



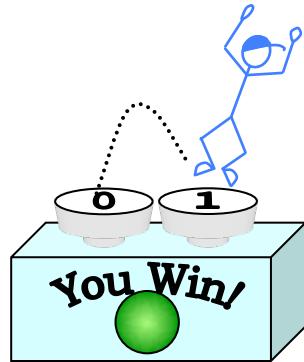
2. Same question:



3. Here's an FSM. Can you discover its rules?

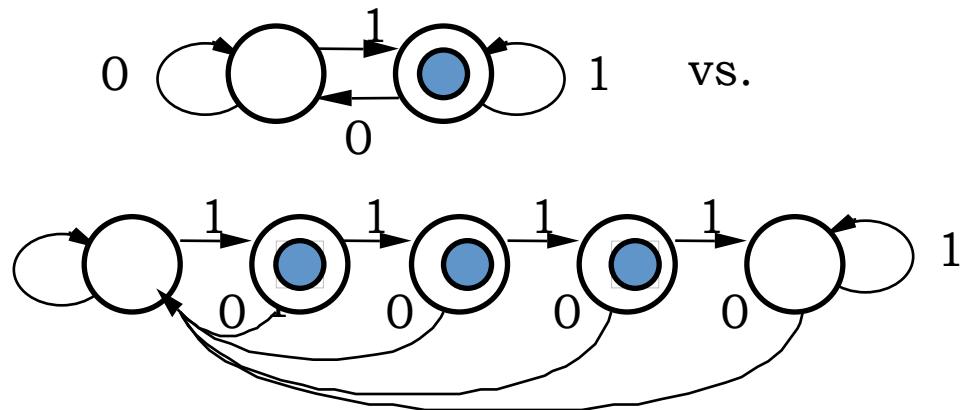


What's My Transition Diagram?



0=OFF,
1=ON?

"1111" = 0
Surprise!

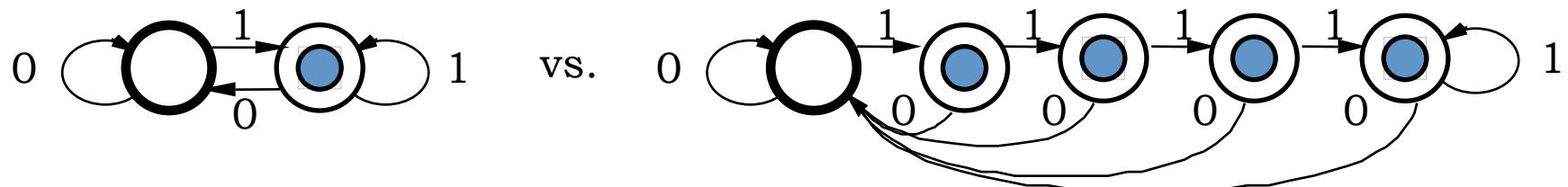


- If you know **NOTHING** about the FSM, you're never sure!
- If you have a **BOUND** on the number of states, you can discover its behavior:

K-state FSM: Every (reachable) state can be reached in $< k$ steps.

BUT ... FSMs may be **equivalent**!

FSM Equivalence



ARE THEY DIFFERENT?

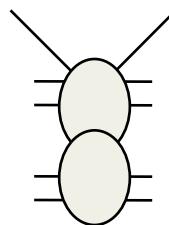
NOT in any practical sense! They are EXTERNALLY INDISTINGUISHABLE, hence interchangeable.

FSMs are *EQUIVALENT* iff every input sequence yields identical output sequences.

ENGINEERING GOAL:

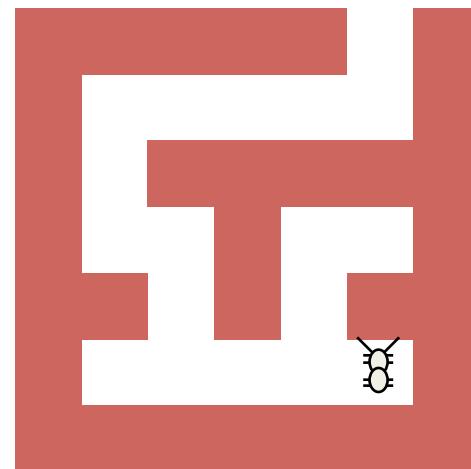
- HAVE an FSM which *works*...
- WANT simplest (ergo cheapest) equivalent FSM.

Let's Build a RoboAnt



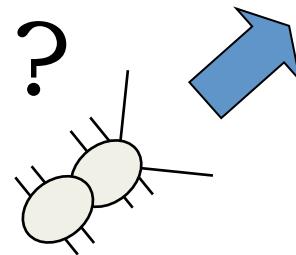
- SENSORS: antennae L and R, each 1 if in contact with something.
- ACTUATORS: Forward Step F, ten-degree turns TL and TR (left, right).

GOAL: Make our ant smart enough to get out of a maze like:

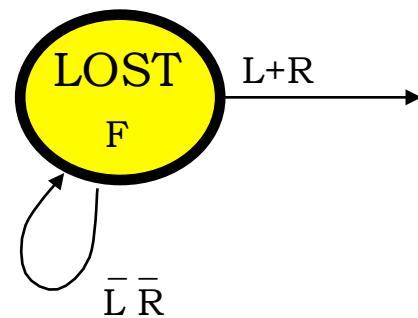


STRATEGY: "Right antenna to the wall"

Lost In Space

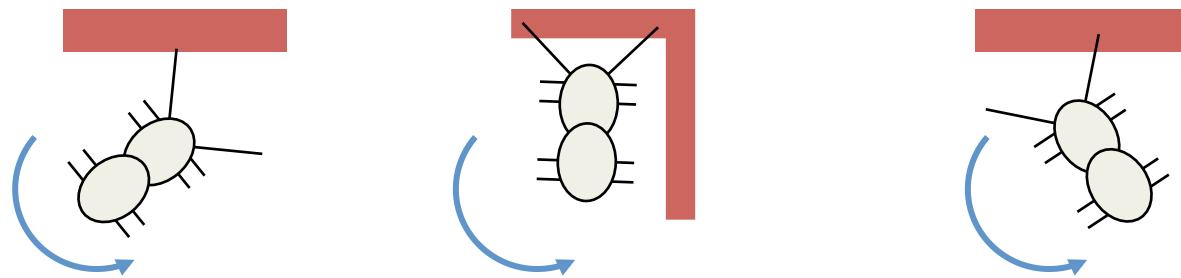


Action: Go forward until we hit something.

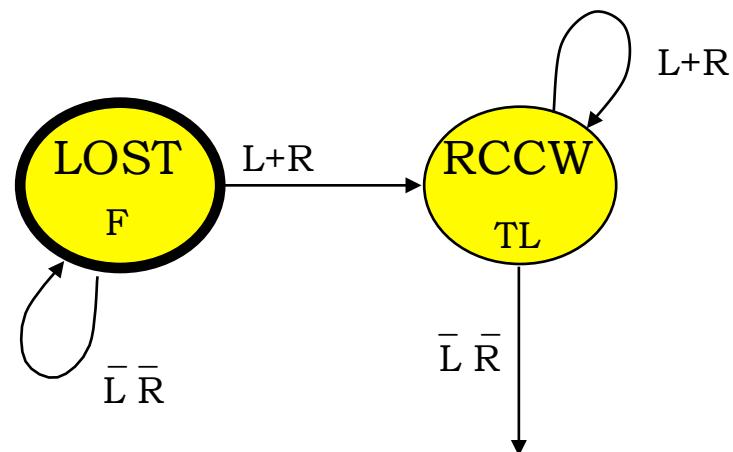


“lost” is the
initial state

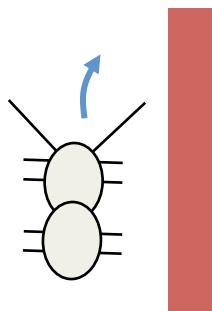
Bonk!



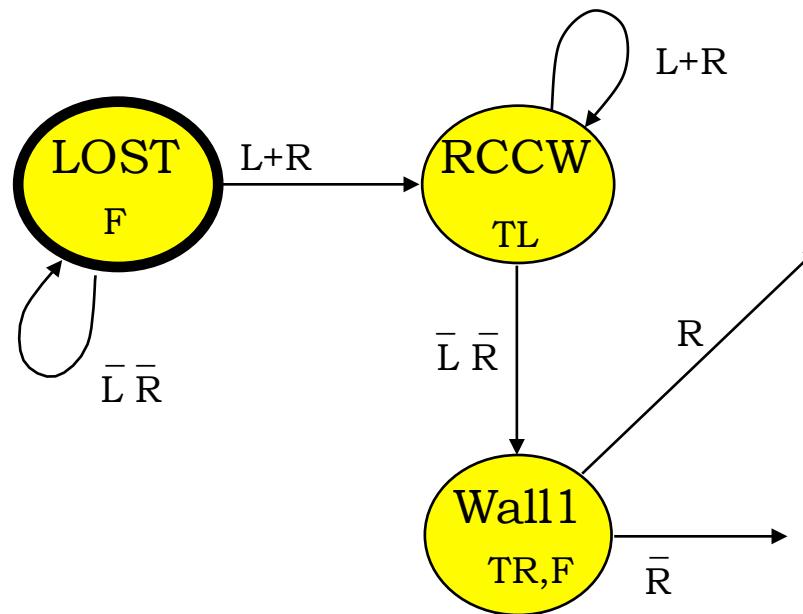
Action: Turn left (CCW) until we don't touch anymore



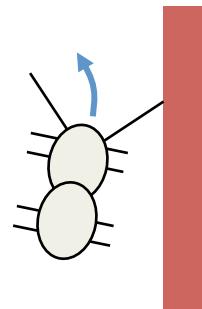
A Little to the Right...



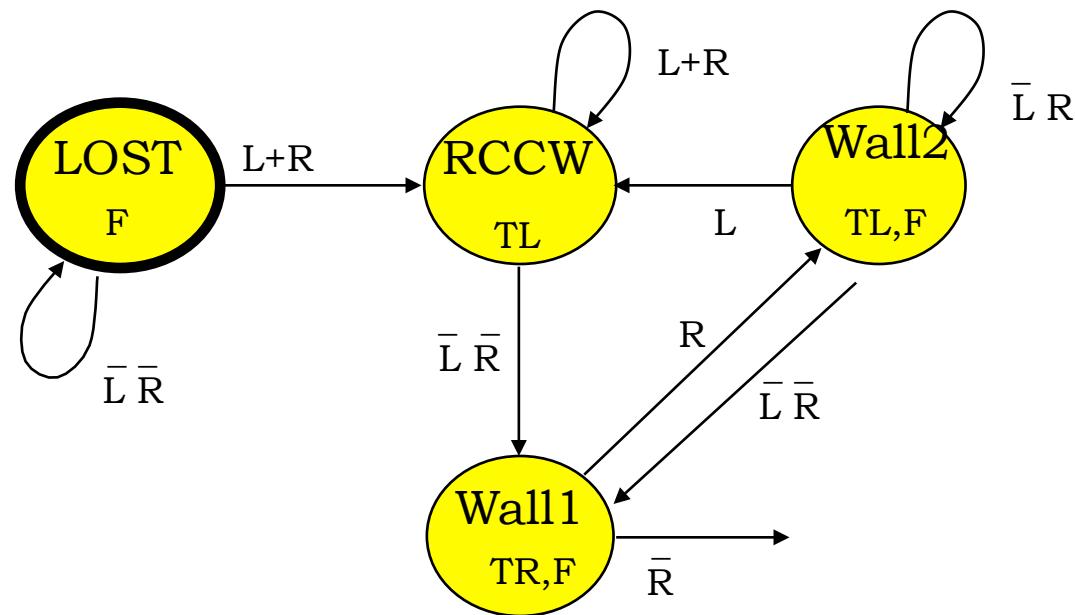
Action: Step and turn right a little, look for wall



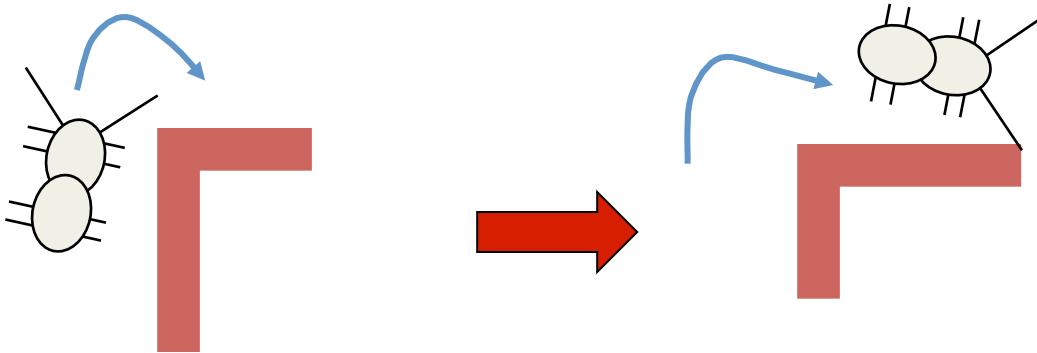
Then a Little to the Left



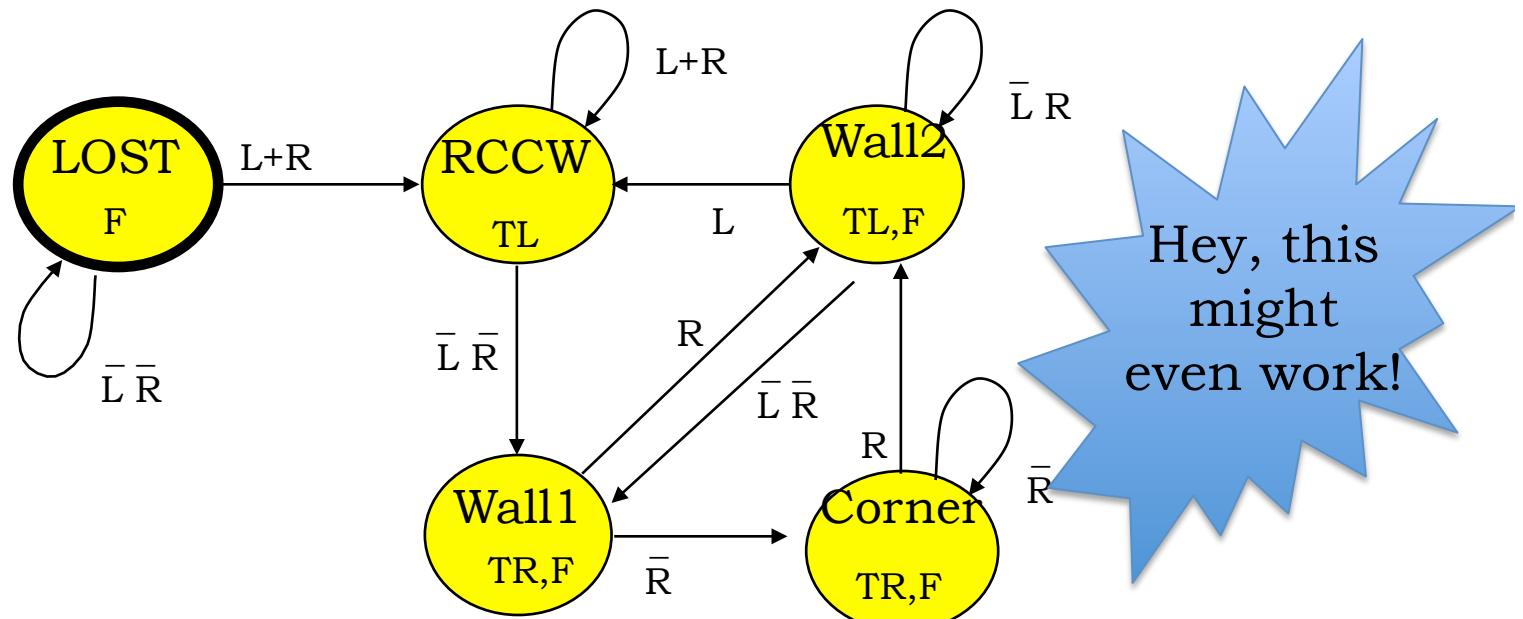
Action: Step and turn left a little, till not touching (again)



Dealing With Outside Corners



Action: Step and turn right until we hit perpendicular wall



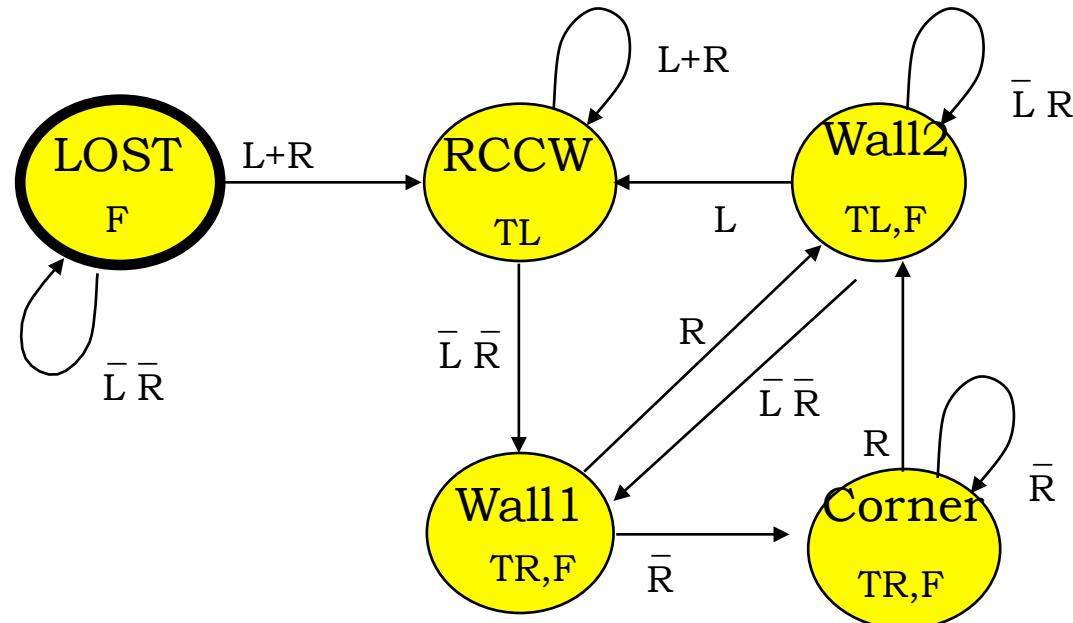
Equivalent State Reduction

Observation: two states are equivalent if

1. Both states have identical outputs; AND
2. Every input \Rightarrow equivalent states.

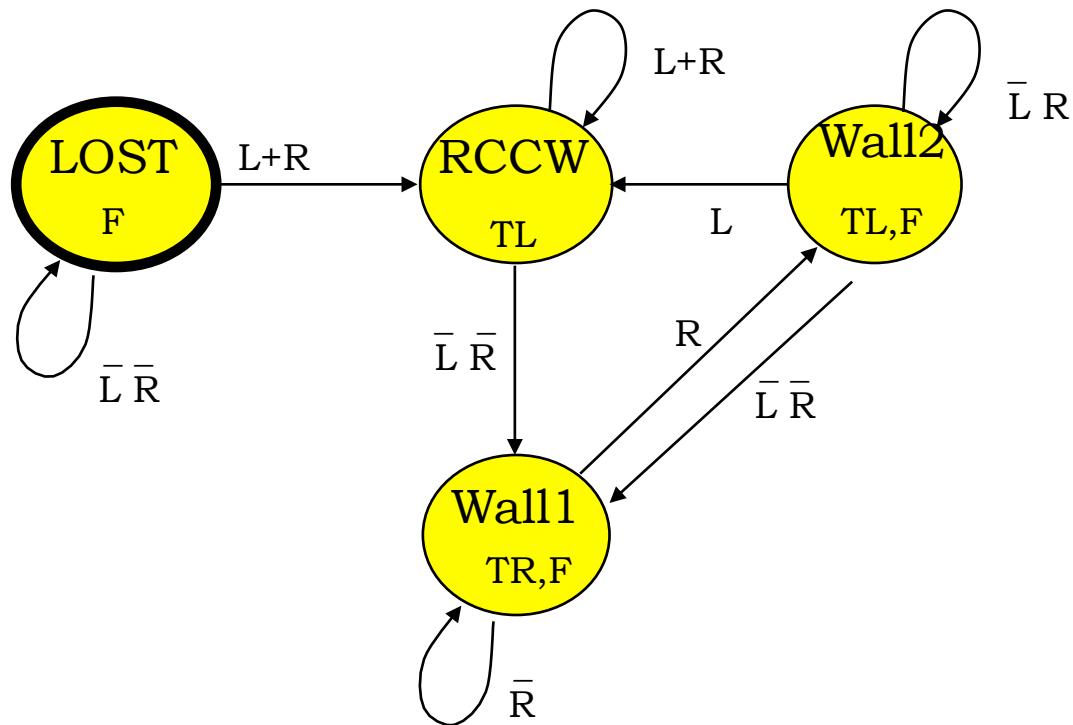
Reduction Strategy:

Find pairs of equivalent states, MERGE them.



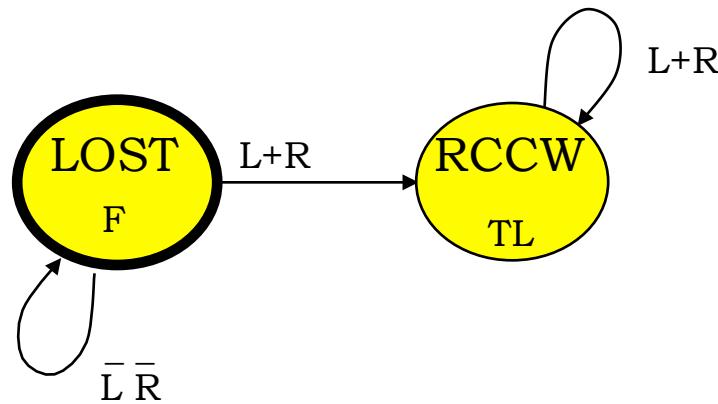
An Evolutionary Step

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!

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

Implementation Details

	S	L	R		S'	TR	TL	F
LOST	00	0	0		00	0	0	1
	00	1	-		01	0	0	1
	00	-	1		01	0	0	1
RCCW	01	1	-		01	0	1	0
	01	-	1		01	0	1	0
	01	0	0		10	0	1	0
WALL1	10	-	0		10	1	0	1
	10	-	1		11	1	0	1
	11	1	-		01	0	1	1
WALL2	11	0	0		10	0	1	1
	11	0	1		11	0	1	1

Complete Transition table

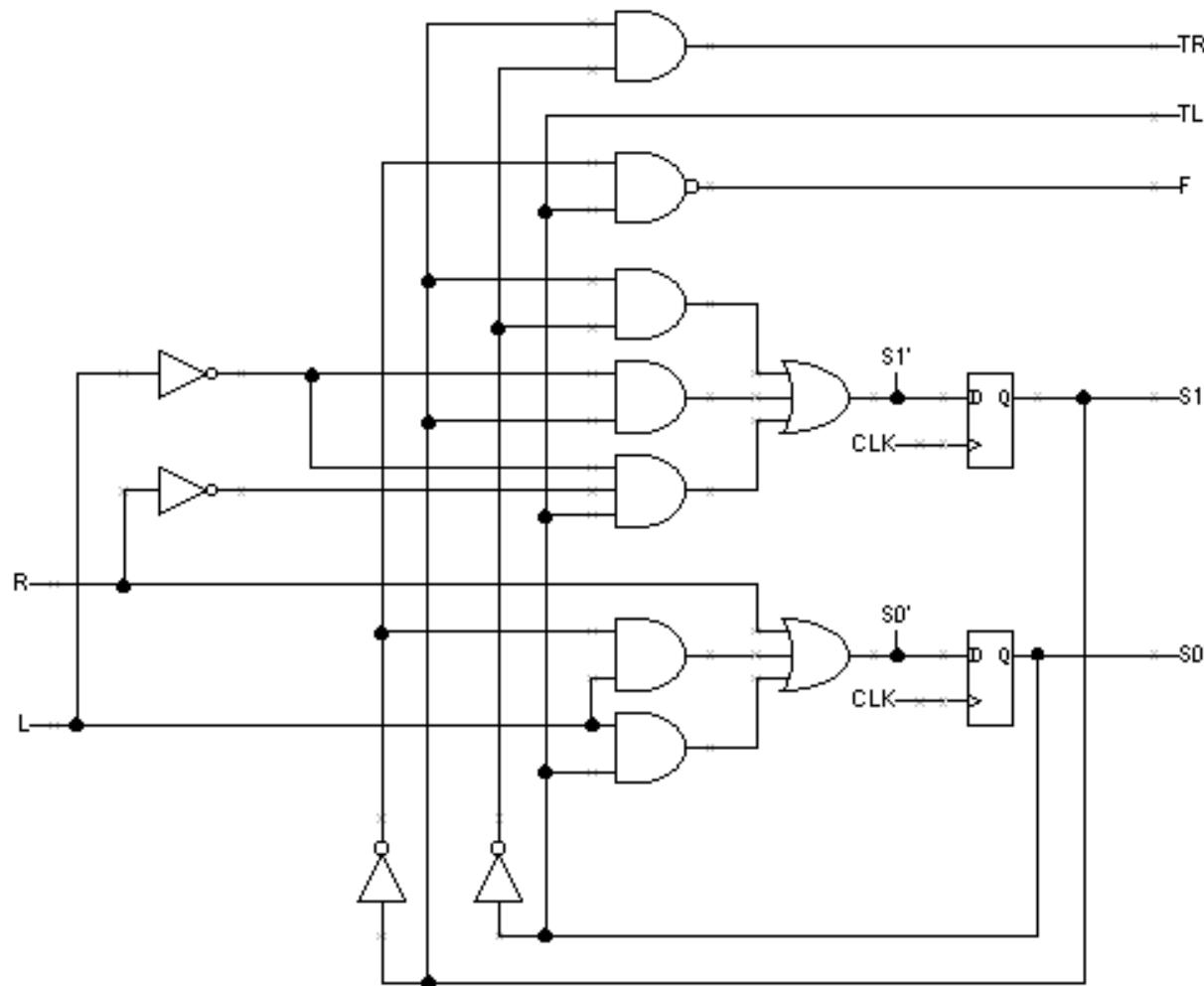
$$S_1' = S_1 \overline{S_0} + \overline{L}S_1 + \overline{LR}S_0$$

	s_1'	$s_1 s_0$
LR	00 01 11 10	00 01 11 10
	00 0 1 1 1	00 1 1 1 1
	01 0 0 1 1	01 0 0 0 1
	11 0 0 0 1	11 0 0 0 1
	10 0 0 0 1	10 0 0 0 1

$$S_0' = R + L\overline{S_1} + LS_0$$

	s_0'	$s_1 s_0$
LR	00 01 11 10	00 01 11 10
	00 0 0 0 0	00 0 0 0 0
	01 1 1 1 1	01 1 1 1 1
	11 1 1 1 1	11 1 1 1 1
	10 1 1 1 0	10 1 1 1 0

Ant Schematic



FSMs All the Way Down?

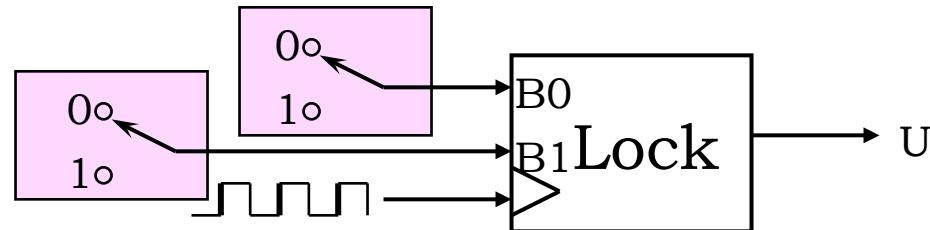
- More than ants:
Swarming, flocking, and schooling can result from collections of very simple FSMs
- Perhaps most physics:
Cellular automata, arrays of simple FSMs, can more accurately model fluids than numerical solutions to PDEs
- What if:
We replaced the ROM with a RAM and have outputs that modify the RAM?

... You'll see FSMs for the rest of your life!

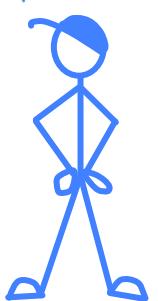


The World Doesn't Run on Our Clock!

What if each button input is an asynchronous 0/1 level?

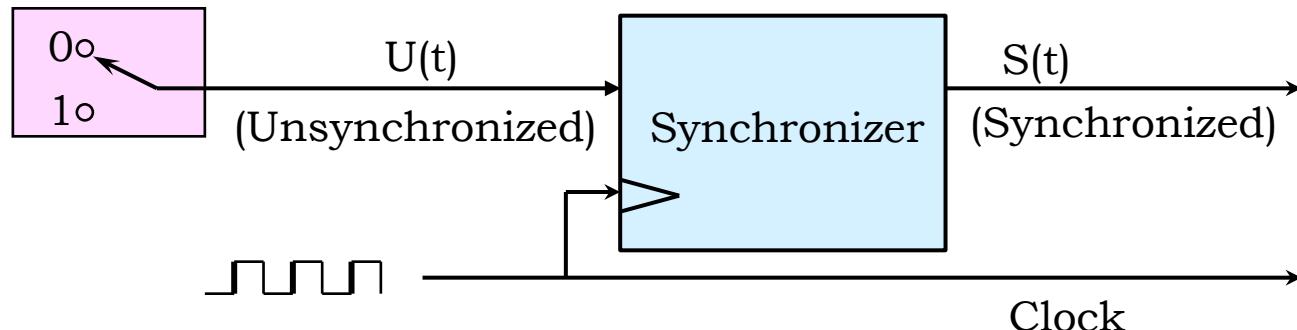


But what about the dynamic discipline?



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:

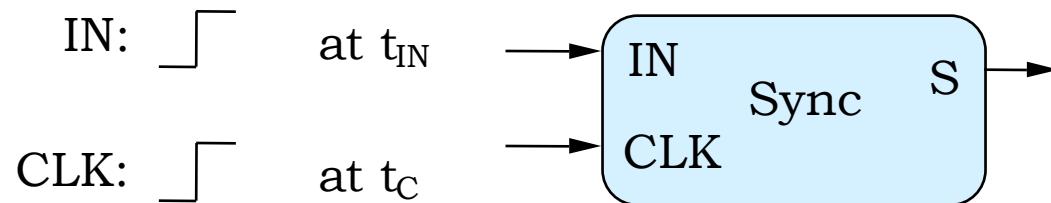


Valid except for brief periods following active clock edges

The Bounded-time Synchronizer

A classic problem

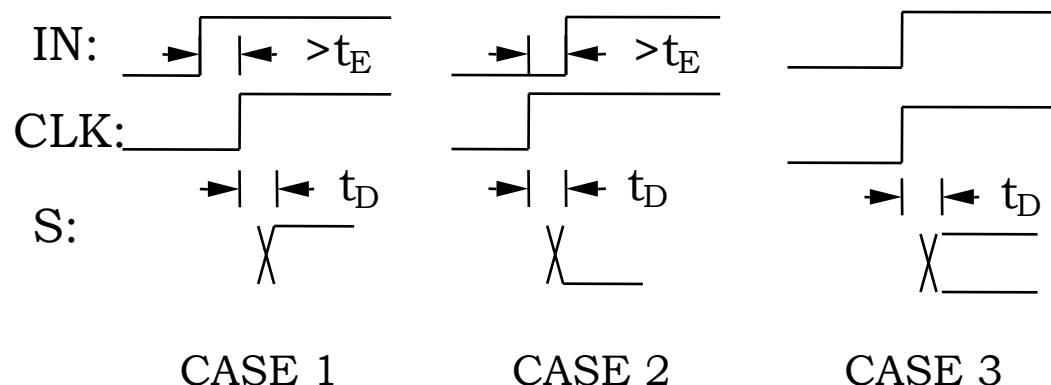
UNSOLVABLE



For NO finite values of t_E and t_D is this spec realizable, even with reliable components!

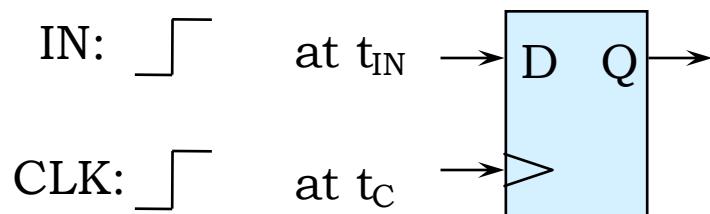
Synchronizer specifications:

- finite t_D (decision time)
- finite t_E (allowable error)
- value of S at time $t_C + t_D$:
 - 1 if $t_{IN} < t_C - t_E$
 - 0 if $t_{IN} > t_C + t_E$
 - 0, 1 otherwise



Unsolvable? That can't be true...

Let's just use a D Register:



We're lured 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 $\max(t_{SETUP}, t_{HOLD})$

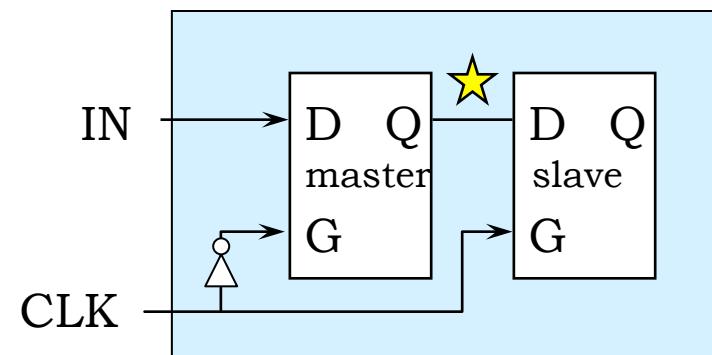
Our logic:

T_{PD} after T_C , we'll have

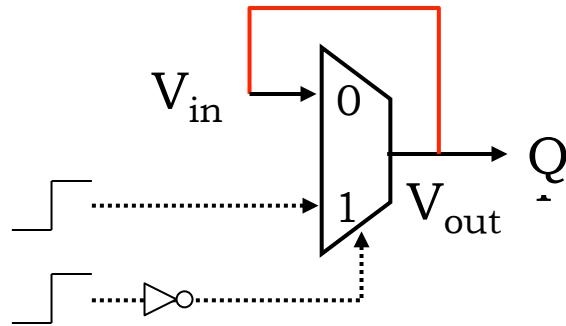
$$Q=1 \text{ iff } t_{IN} + t_{SETUP} < t_C$$

$$Q=0 \text{ iff } t_C + t_{HOLD} < t_{IN}$$

$Q=0$ or 1 otherwise.

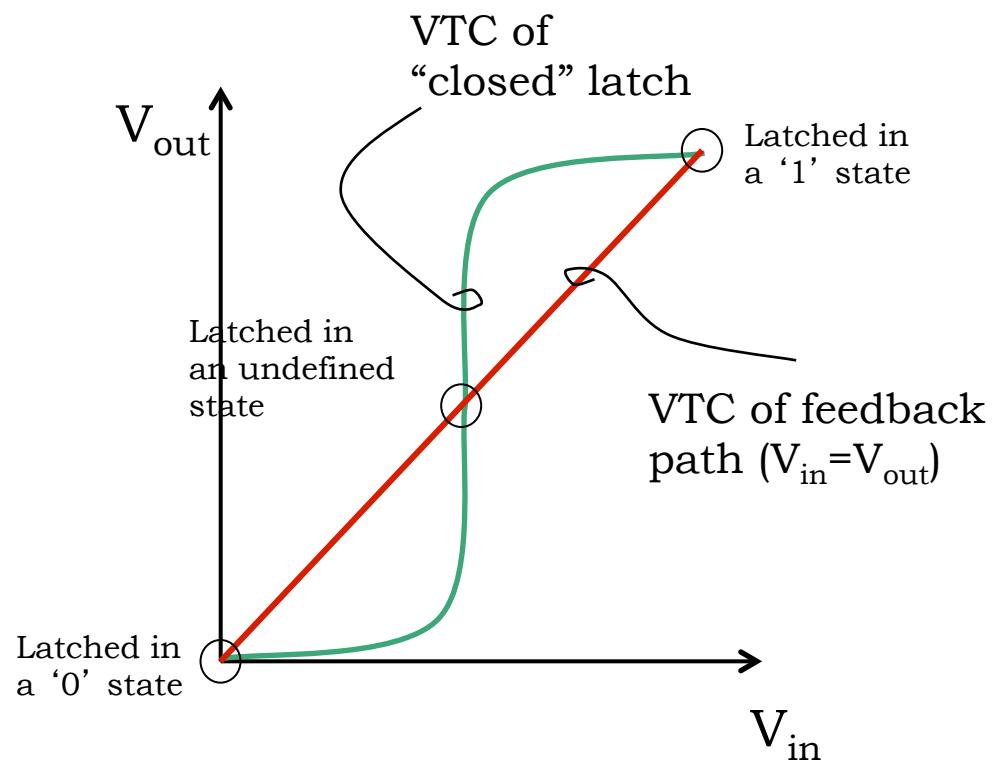


The Mysterious Metastable State



Recall that the latch output is the solution to two simultaneous constraints:

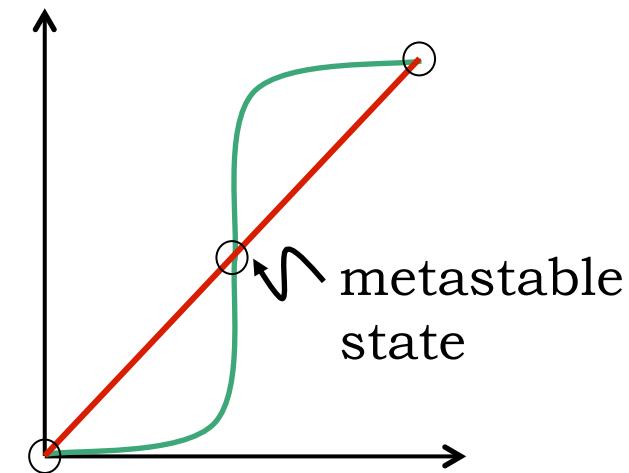
1. The VTC of path thru MUX; and
2. $V_{in} = V_{out}$



In addition to our expected stable solutions, we find an unstable equilibrium in the forbidden zone called the “Metastable State”

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!



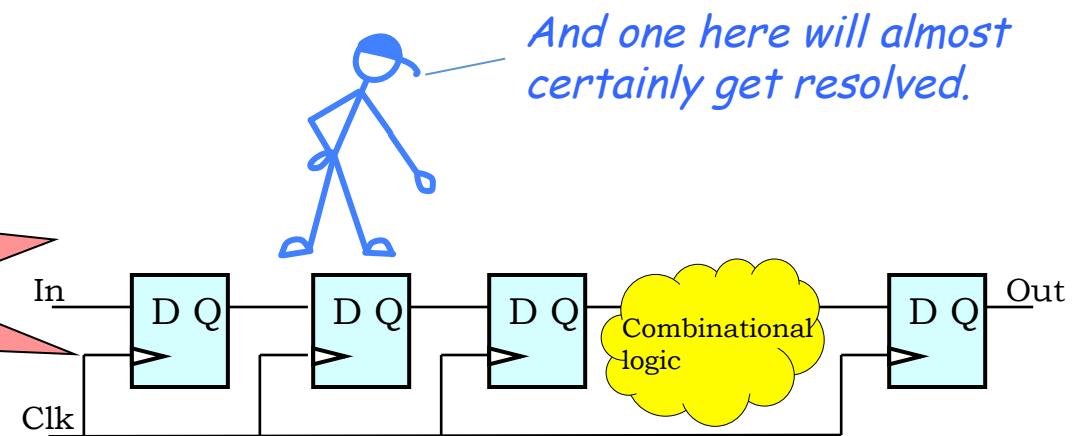
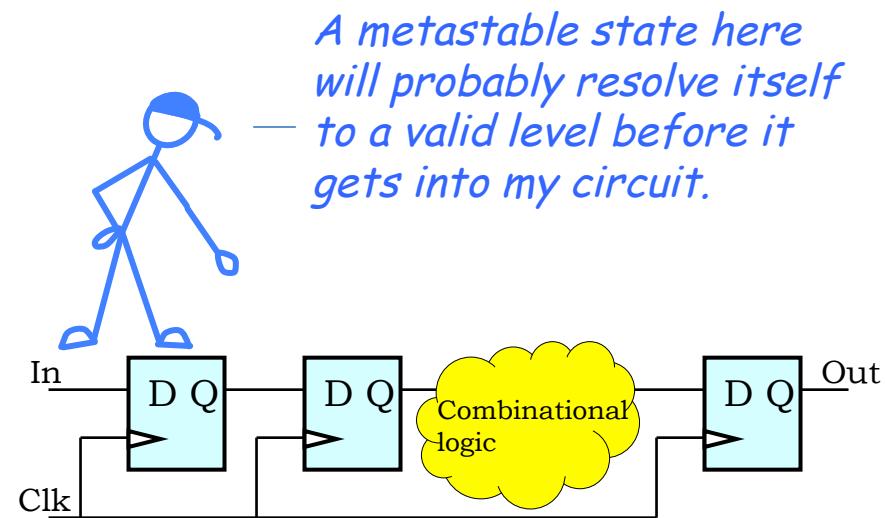
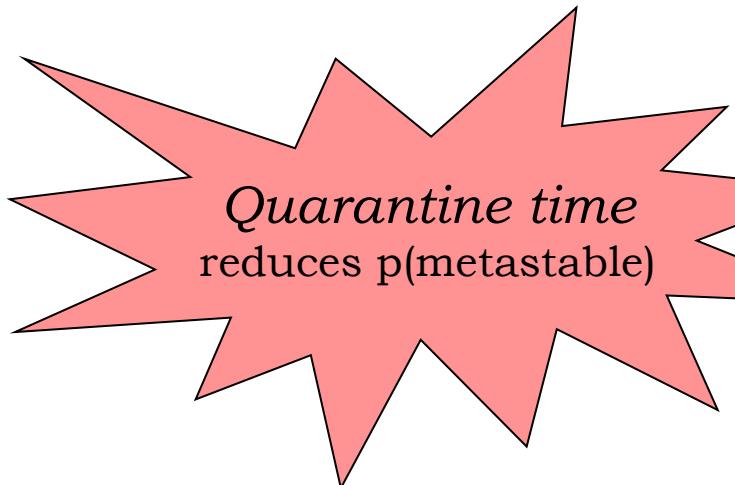
If metastable at t_0 :

- $p(\text{metastable at } t_0+T) > 0$ for finite T
- $p(\text{metastable at } t_0+T)$ decreases exponentially with increasing T

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.



7. Performance Measures

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS

Forget circuits... Let's Solve a Real Problem

INPUT:
dirty laundry



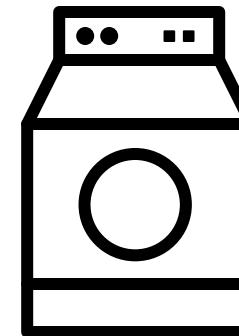
OUTPUT:
6 more weeks



Device: Washer

Function: Fill, Agitate, Spin

$\text{Washer}_{\text{PD}} = 30 \text{ mins}$



Device: Dryer

Function: Heat, Spin

$\text{Dryer}_{\text{PD}} = 60 \text{ mins}$

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:



$$\begin{aligned}\text{Total}_{\text{PD}} &= \text{Washer}_{\text{PD}} + \text{Dryer}_{\text{PD}} \\ &= \underline{\hspace{2cm} 90 \hspace{2cm}} \text{ mins}\end{aligned}$$

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:



...

$$\text{Total}_{\text{PD}} = N * (\text{Washer}_{\text{PD}} + \text{Dryer}_{\text{PD}})$$

$$= \frac{\text{N*90}}{} \text{ mins}$$

Doing N Loads... The 6.004 Way

6.004 students
“pipeline” the laundry
process.

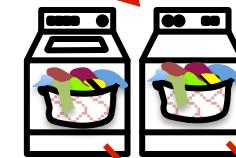
That's why we wait!

Actually, it's more like $N \cdot 60 + 30$ 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.

Step 1:



Step 2:



Step 3:

...

$$\text{Total}_{\text{PD}} = N * \text{Max}(\text{Washer}_{\text{PD}}, \text{Dryer}_{\text{PD}})$$

$$= \underline{\text{N}*60} \text{ mins}$$

Performance Measures

Latency:

The delay from when an input is established until the output associated with that input becomes valid.

$$\text{Harvard Laundry} = \frac{90}{\text{ }} \text{ mins}$$

$$6.004 \text{ Laundry} = \frac{120}{\text{ }} \text{ mins} \leftarrow$$

Assuming that the wash is started as soon as possible and waits (wet) in the washer until dryer is available.

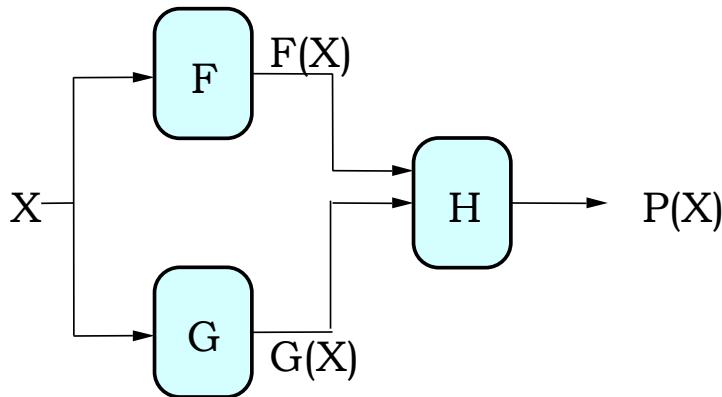
Throughput:

The *rate* at which inputs or outputs are processed.

$$\text{Harvard Laundry} = \frac{1/90}{\text{ }} \text{ outputs/min}$$

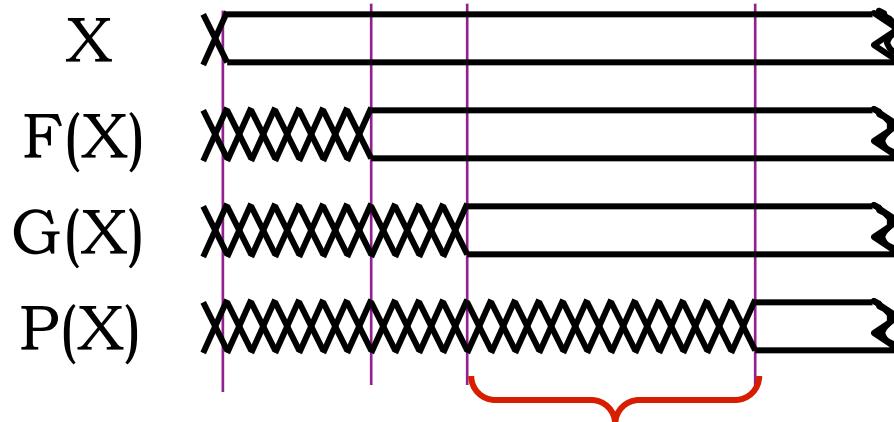
$$6.004 \text{ Laundry} = \frac{1/60}{\text{ }} \text{ outputs/min}$$

Okay, Back To Circuits...

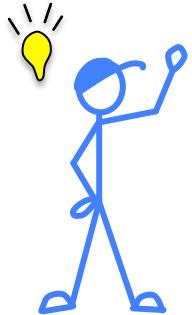


For combinational logic:
latency = t_{PD} ,
throughput = $1/t_{PD}$.

We can't get the answer faster,
but are we making effective use
of our hardware at all times?

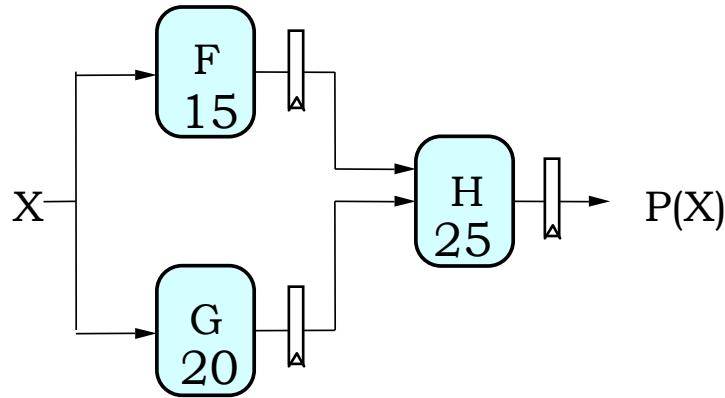


F & G are “idle”, just holding their outputs stable while H performs its computation



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:

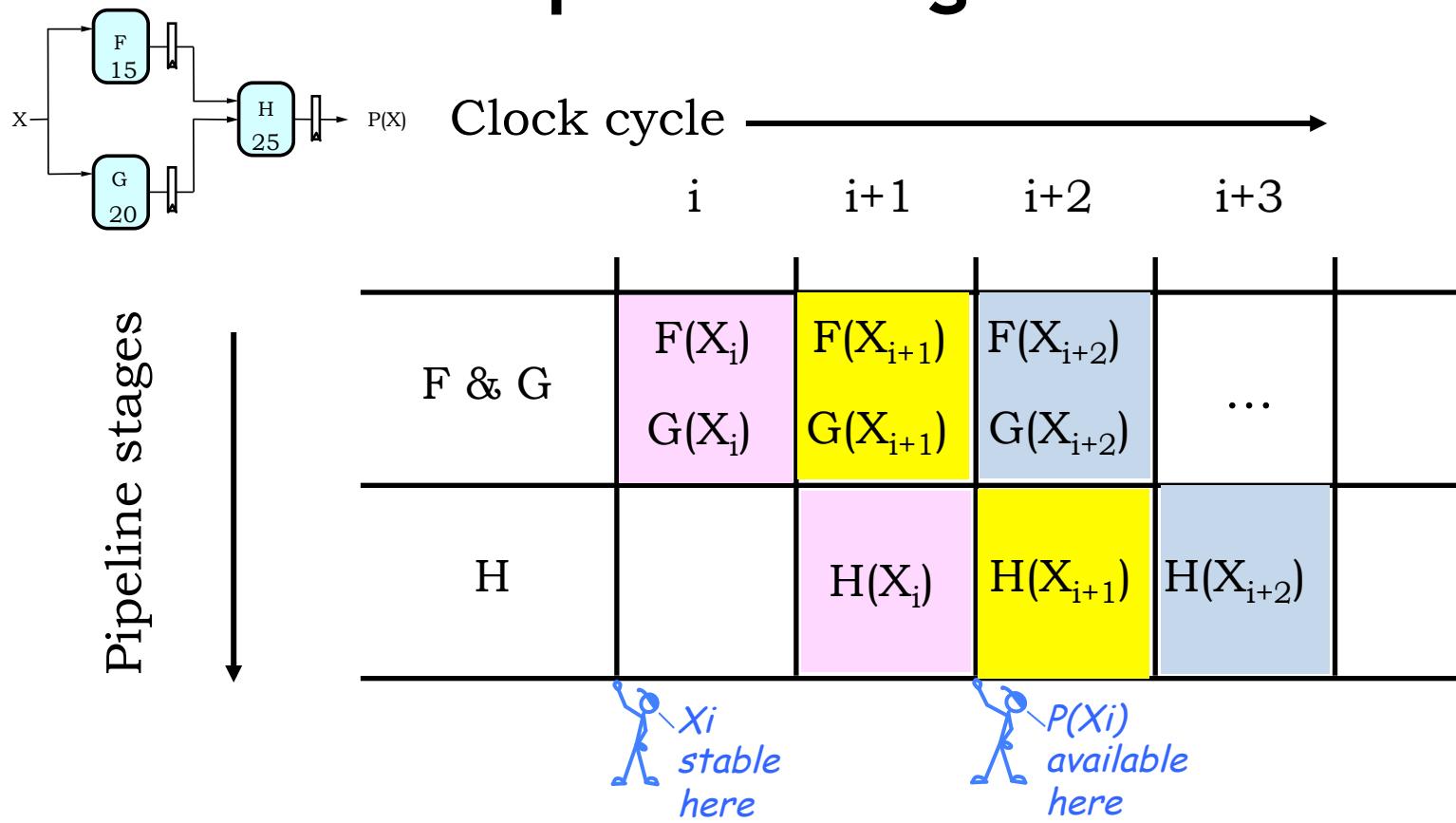
latency throughput

unpipelined	45	1/45
-------------	----	------

2-stage pipeline	<u>50</u>	<u>1/25</u>
------------------	-----------	-------------



Pipeline Diagrams



The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock 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:

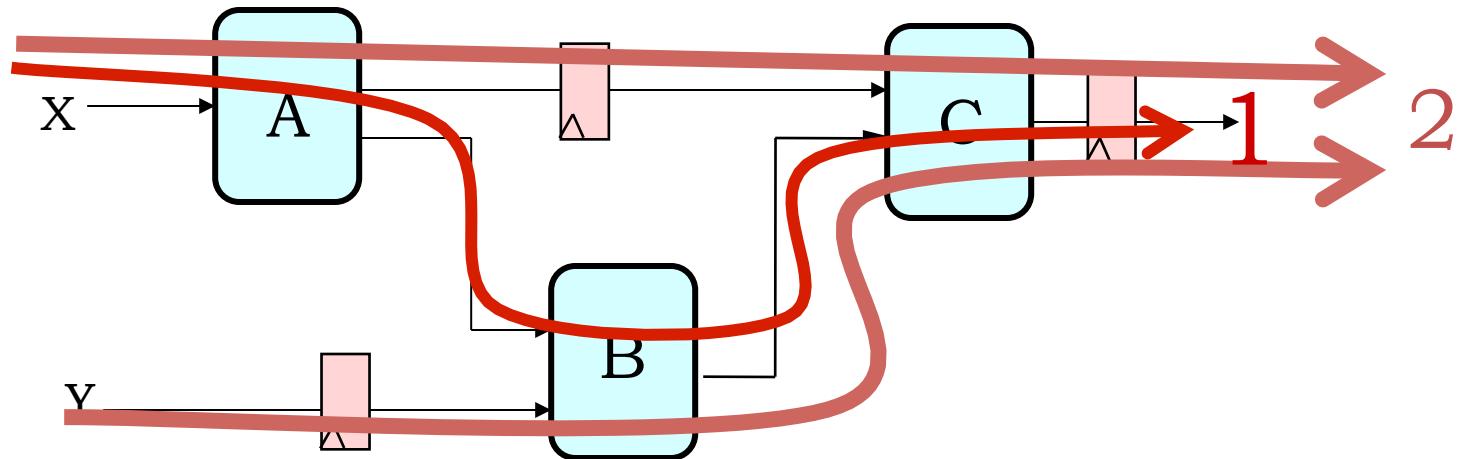
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP} .

The LATENCY of a K-pipeline is K times the period of the system’s clock.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

Ill-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., $B(A(X_{i+1}), Y_i)$. 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!

A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

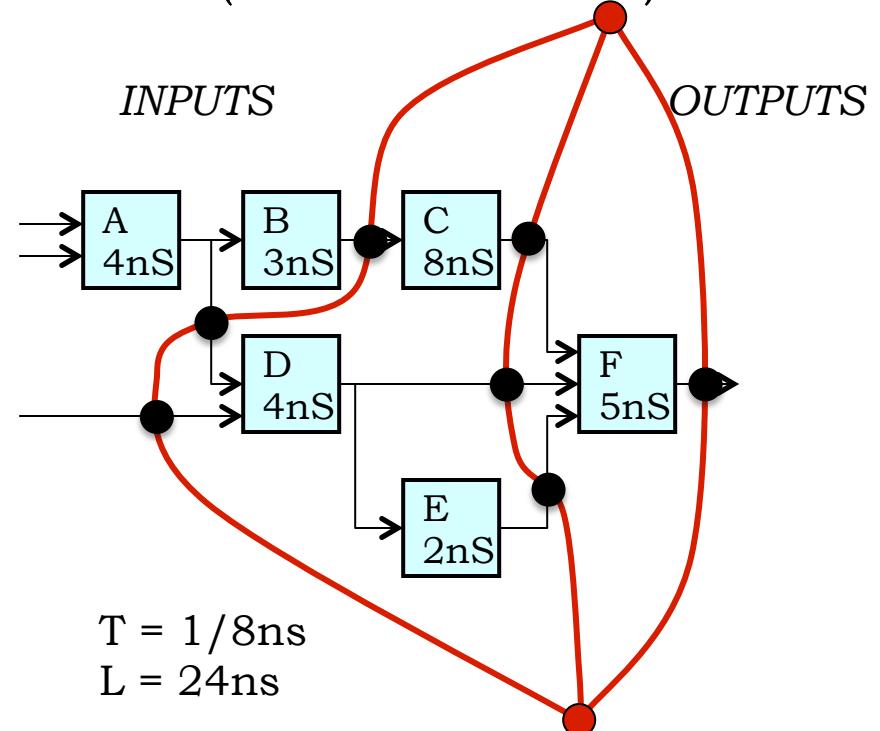
Step 2:

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*.

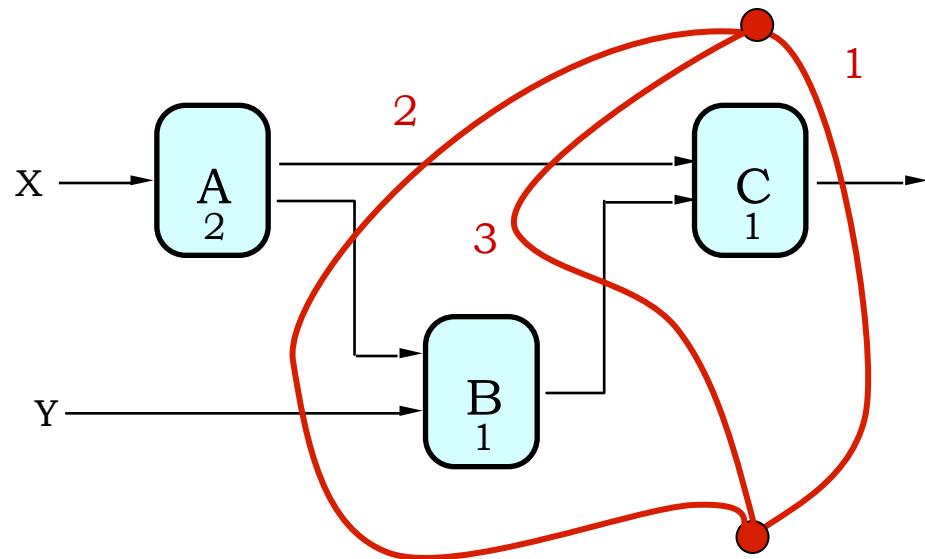
Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



Pipeline Example



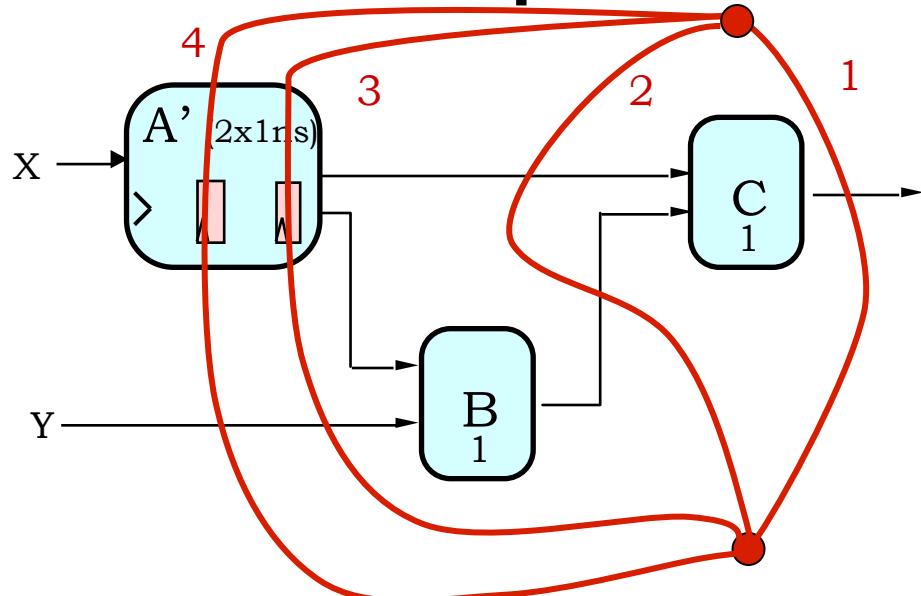
	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

+ increase throughput
- increase latency
- “bottleneck” problem

Pipelined Components



4-stage pipeline, throughput=1

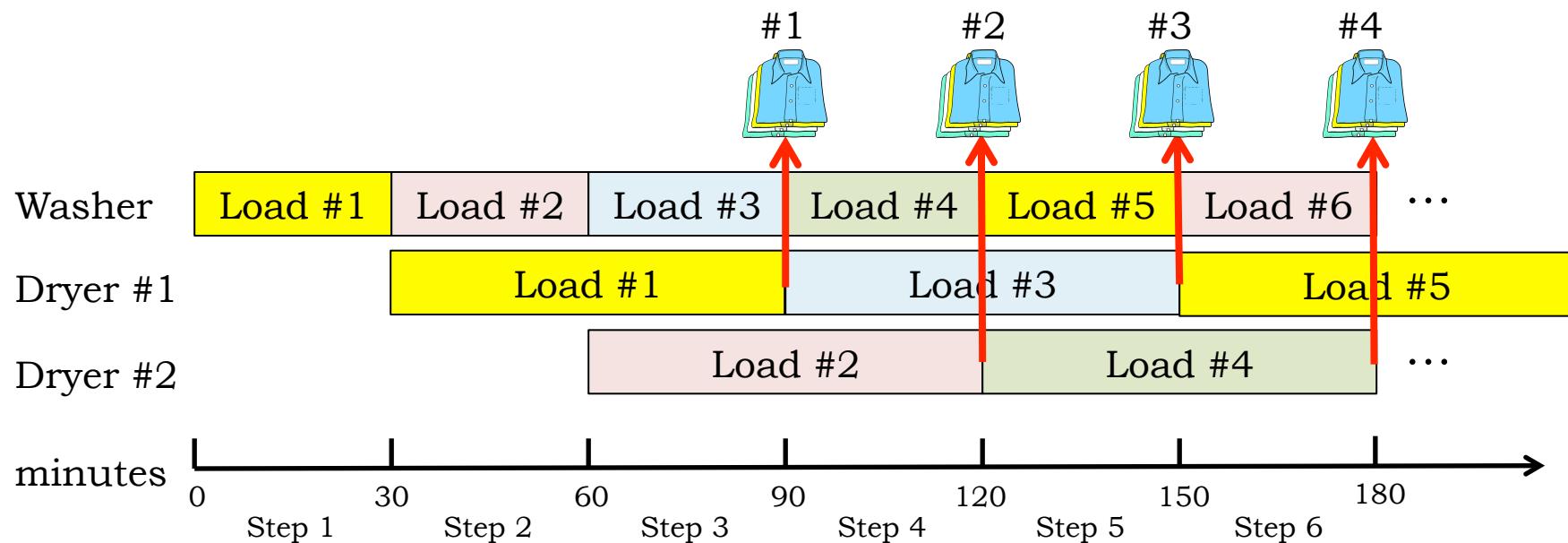
Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may let us decrease the clock period
- Must account for new pipeline stages in our plan



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.



$$\text{Throughput} = \underline{1/30} \text{ loads/min}, \text{Latency} = \underline{90} \text{ mins/load}$$

Back To Our Bottleneck...

Recall our earlier example...

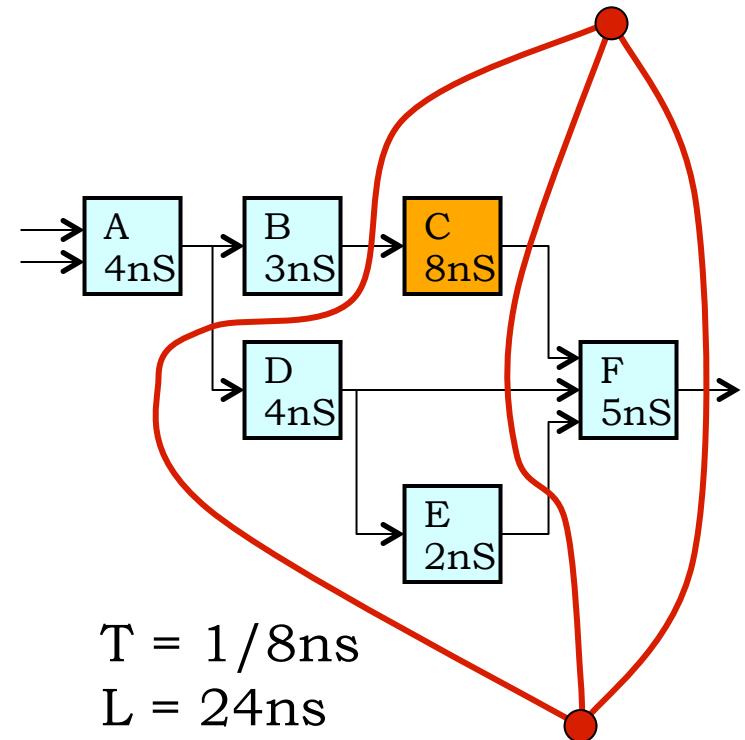
- C – the slowest component – limits clock period to 8 ns.
- HENCE throughput limited to $1/8$ ns.

We could improve throughput by

- Finding a pipelined version of C;

OR ...

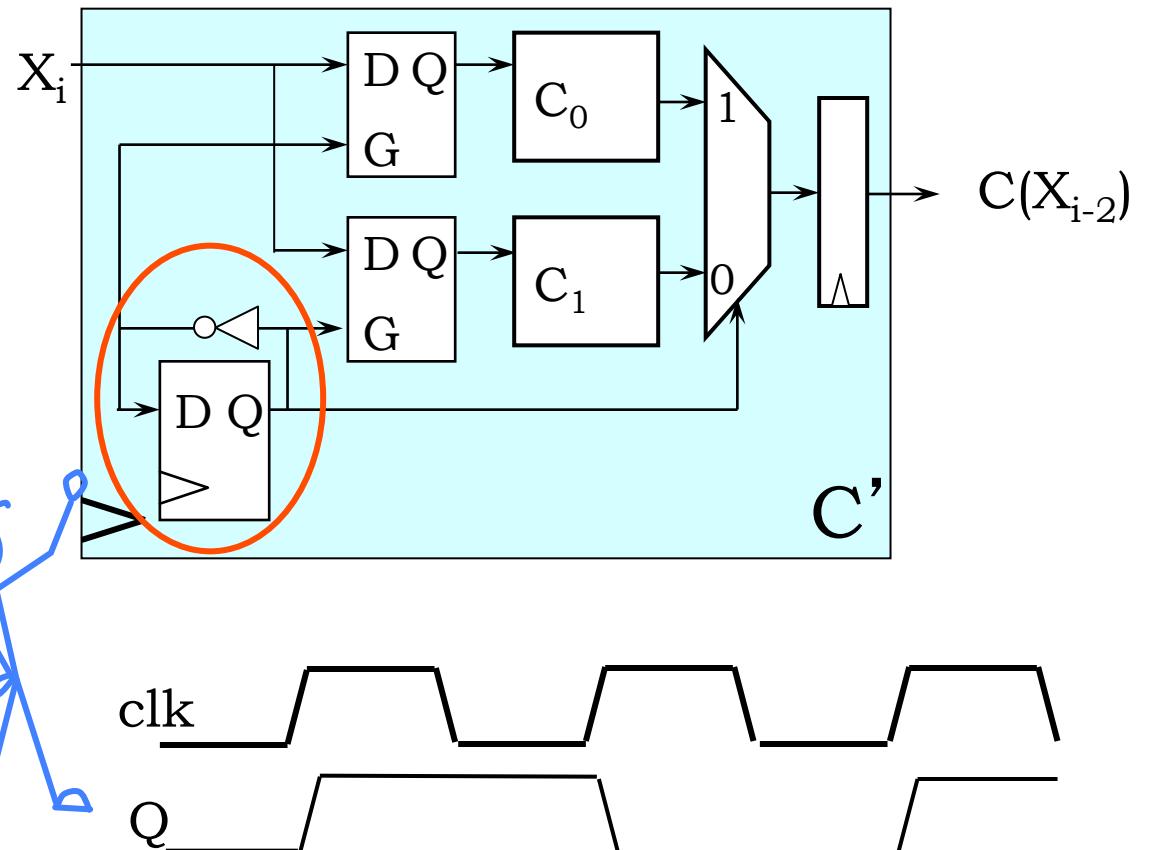
- *interleaving* multiple copies of C!



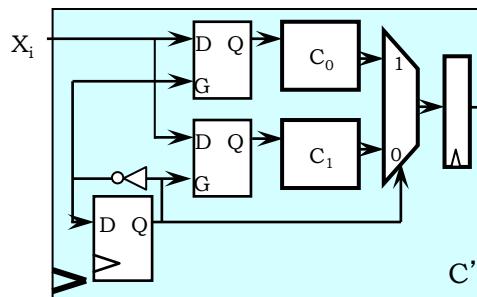
Circuit Interleaving

We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.

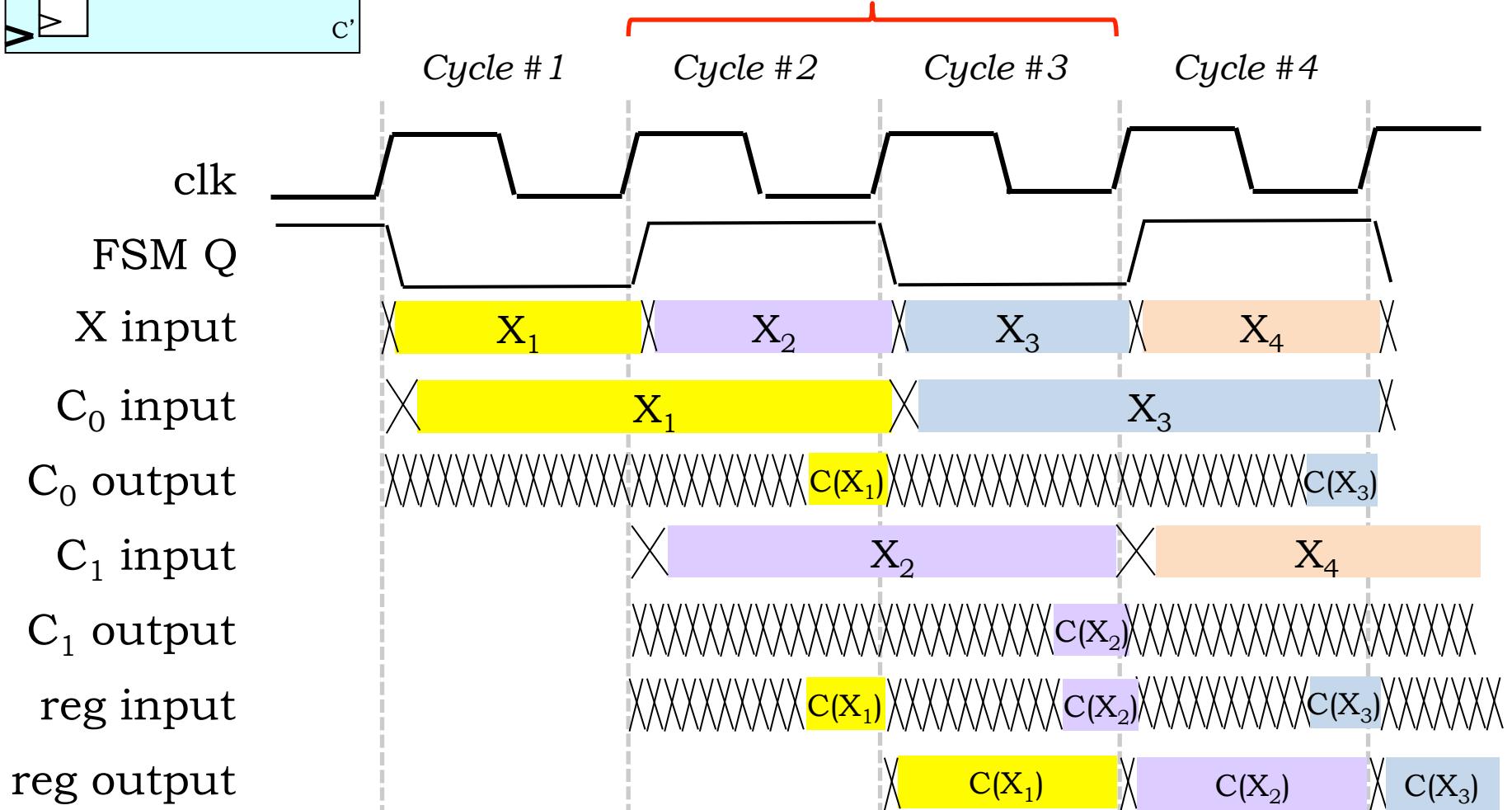
This is a simple 2-state FSM that alternates between 0 and 1 on each clock



Circuit Interleaving



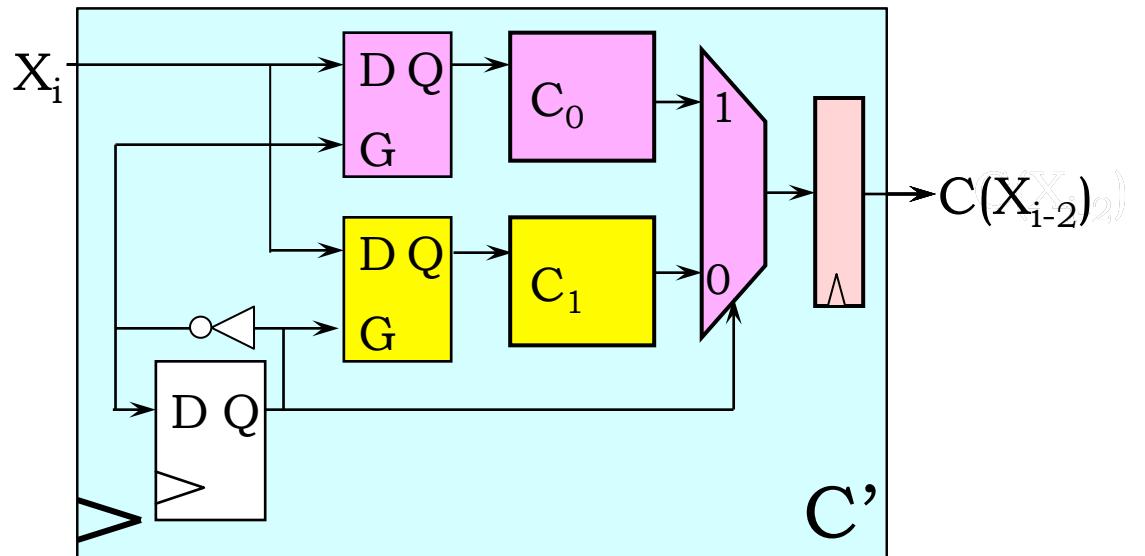
$$2 \cdot t_{CLK} \geq (t_{PD,upstreamREG} + t_{PD,LATCH} + t_{PC,C} + t_{PD,MUX} + t_{SETUP,REG})$$



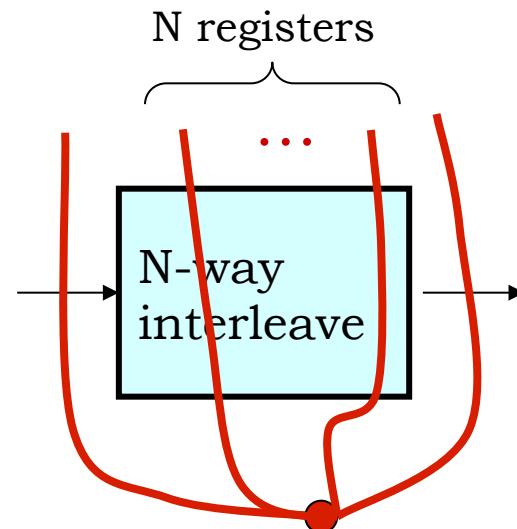
Circuit Interleaving

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...

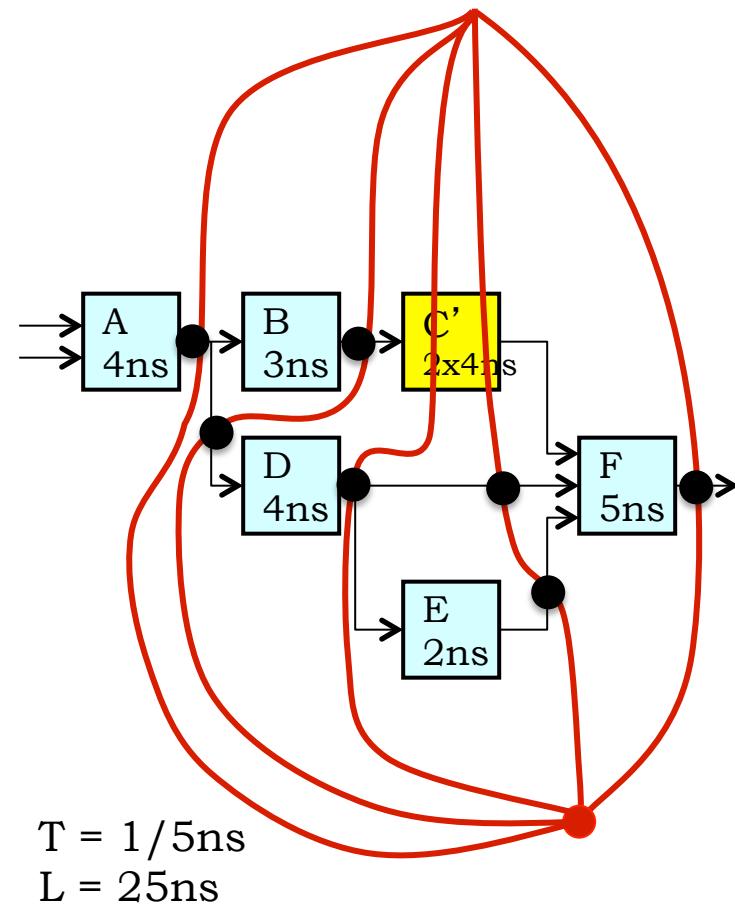


Combine Techniques

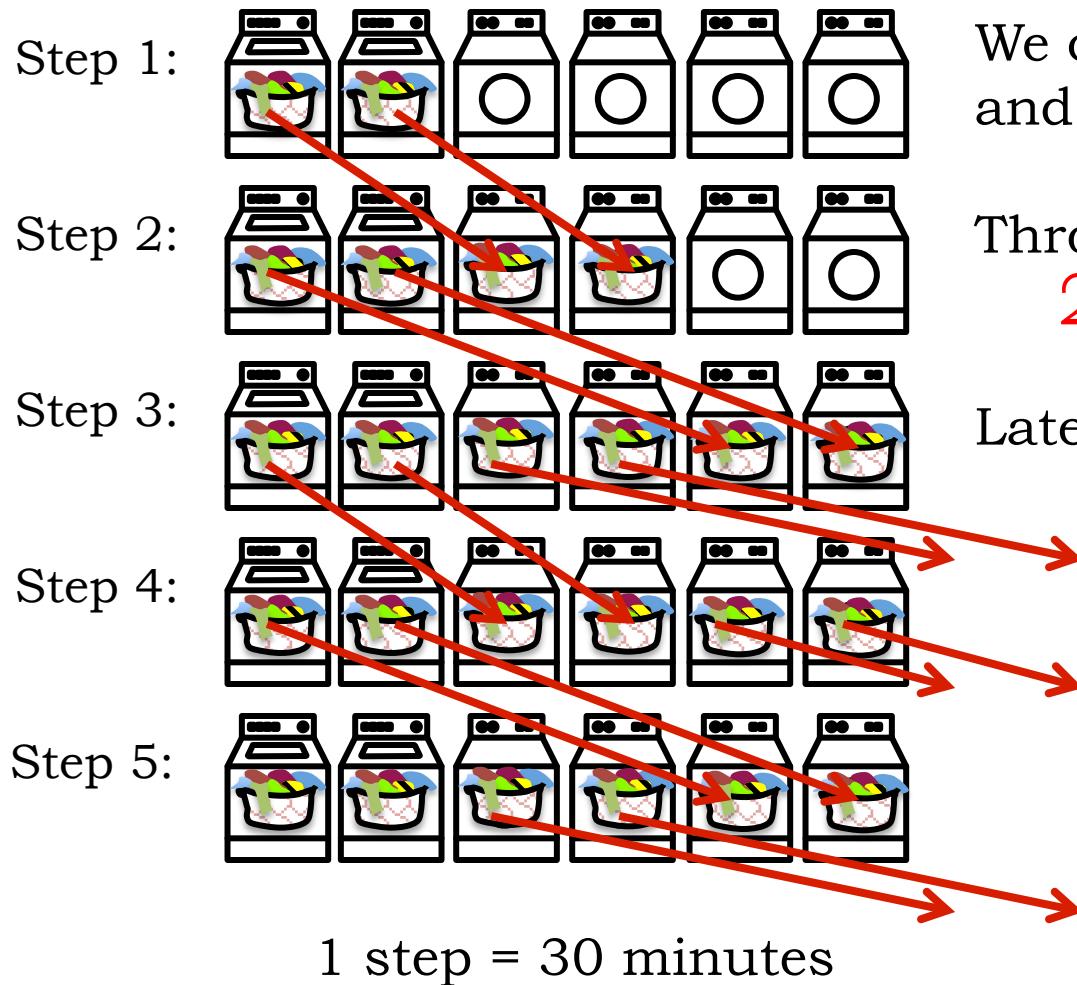
We can combine interleaving and pipelining. Here, C' interleaves two C elements and has an effective tCLK of 4 ns and a latency of 8 ns.

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.



And Add A Little Parallelism...



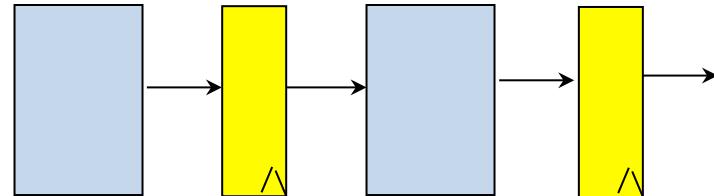
We can combine interleaving and pipelining with parallelism.

Throughput =
 $2/30 = 1/15$ load/min

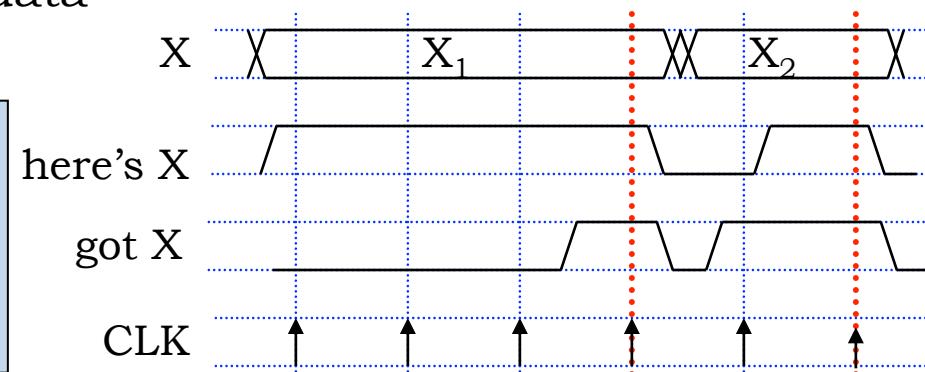
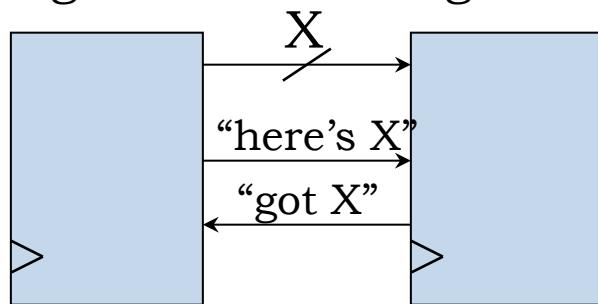
Latency = 90 min

Control Structure Alternatives

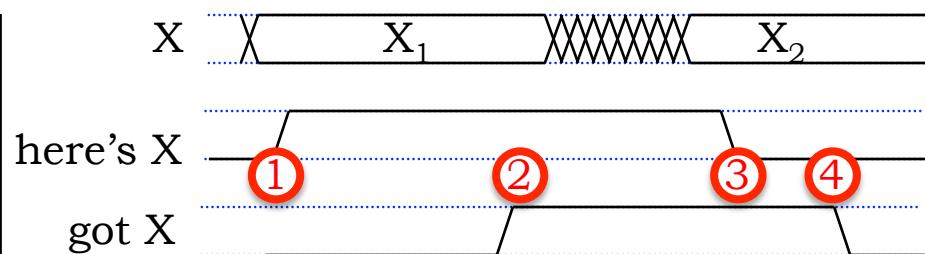
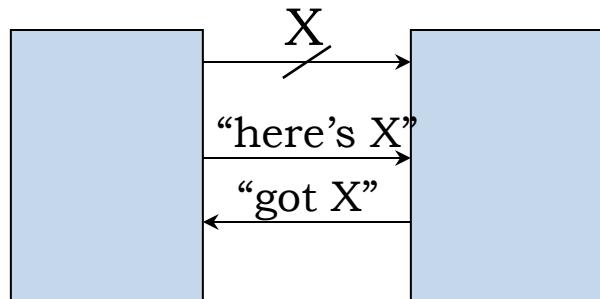
Synchronous, globally-timed:



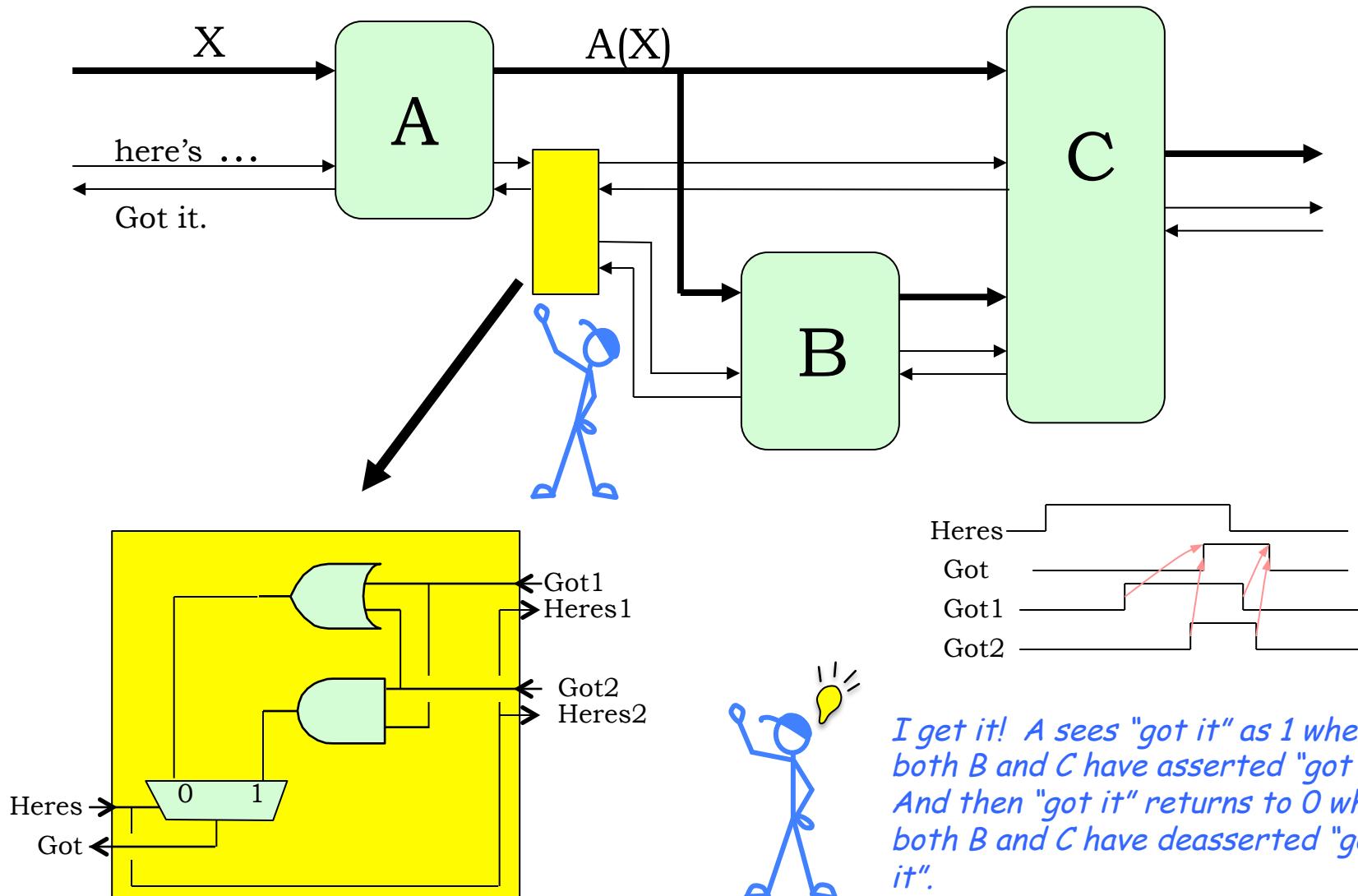
Synchronous, locally-timed:
Local FSMs control flow of data
using “handshake” signals



Asynchronous, locally-timed system using *transition signaling*:

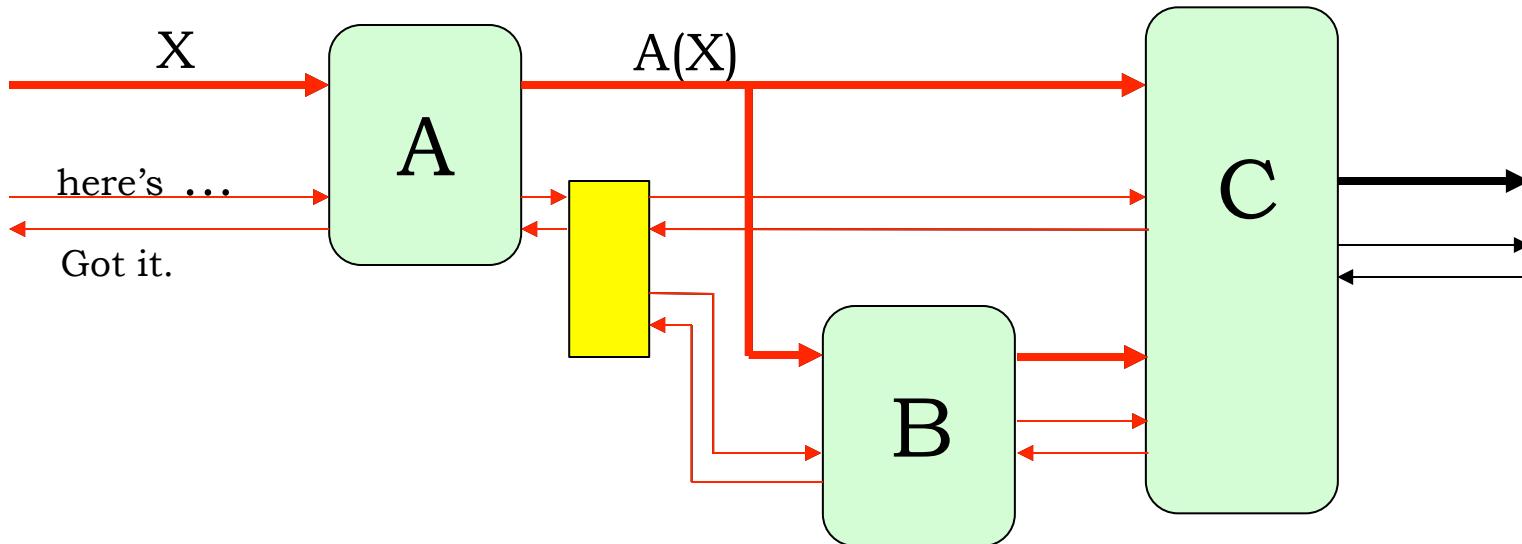


Self-timed Example



I get it! A sees "got it" as 1 when both B and C have asserted "got it". And then "got it" returns to 0 when both B and C have deasserted "got it".

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

Control Structure Taxonomy

Easy to design but fixed-sized interval can be wasteful (no data-dependencies in timing)

Large systems lead to very complicated timing generators... just say no!

Synchronous

Globally Timed

Centralized clocked FSM generates all control signals.

Asynchronous

Locally Timed

Start and Finish signals generated by each major subsystem, synchronously with global clock.

Central control unit tailors current time slice to current tasks.

Each subsystem takes asynchronous Start, generates asynchronous Finish (perhaps using local clock).

The best way to build large systems that have independently-timed components.

The “next big idea” for the last several decades: a lot of design work to do in general, but extra work is worth it in special cases

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

8. Design Tradeoffs

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS

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. ...

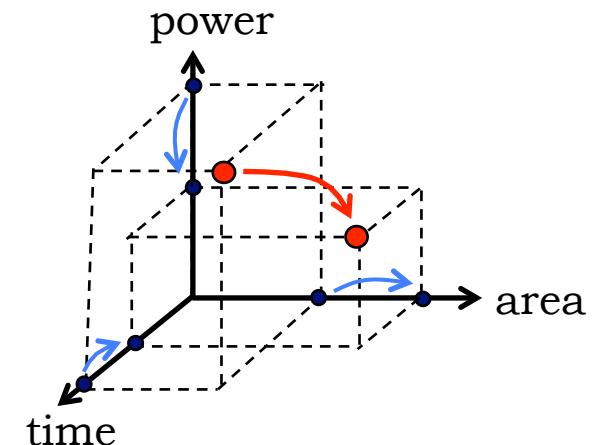


©Advanced Micro Devices (with permission)

VS.

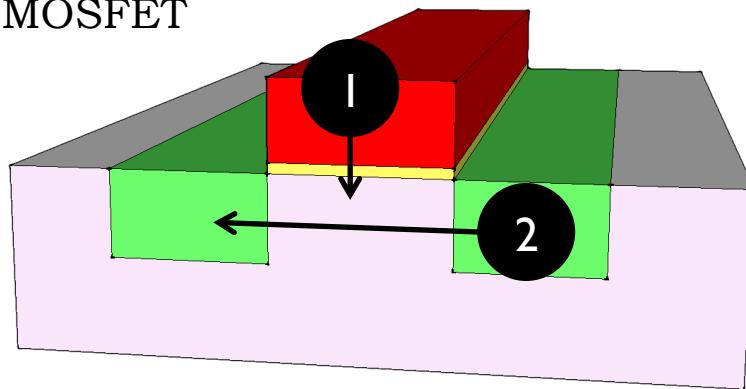


Justin14 (CC BY-SA 4.0)



CMOS Static Power Dissipation

MOSFET



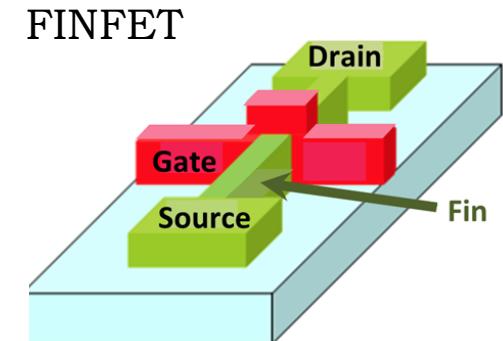
1

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)

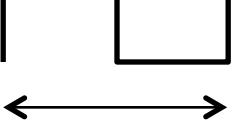
- Leakage gets larger as difference between V_{TH} and “off” gate voltage (eg, V_{OL} in an nfet) gets smaller.
Significant as V_{TH} has become smaller.
- Fix: 3D FINFET wraps gate around inversion region



Irene Ringworm (CC BY-SA 3.0)

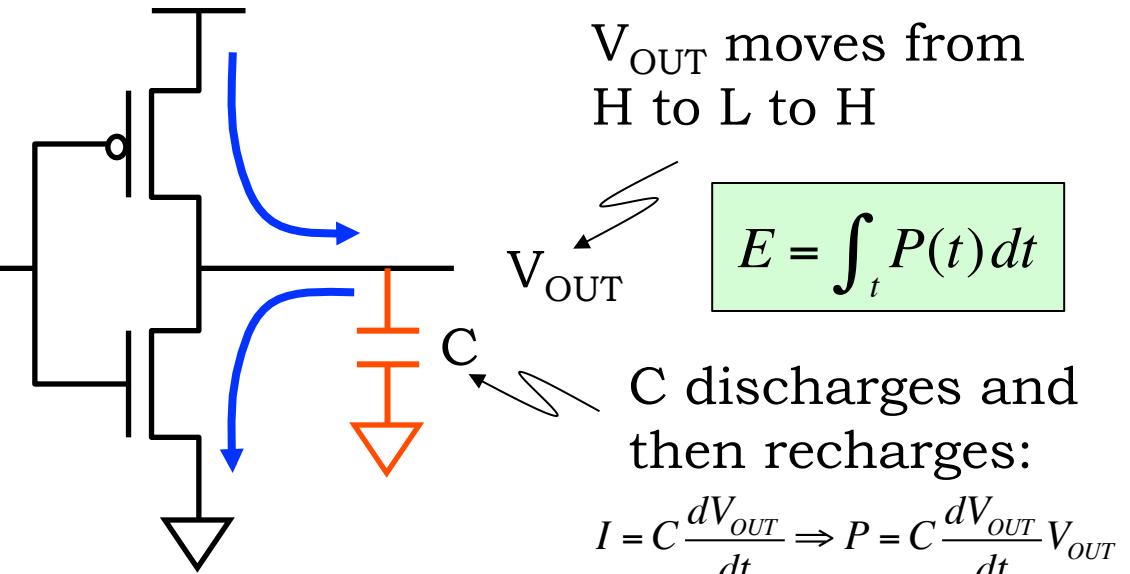
CMOS Dynamic Power Dissipation

V_{IN} moves from L to H to L


 $t_{CLK} = 1/f_{CLK}$

Power dissipated to discharge C:

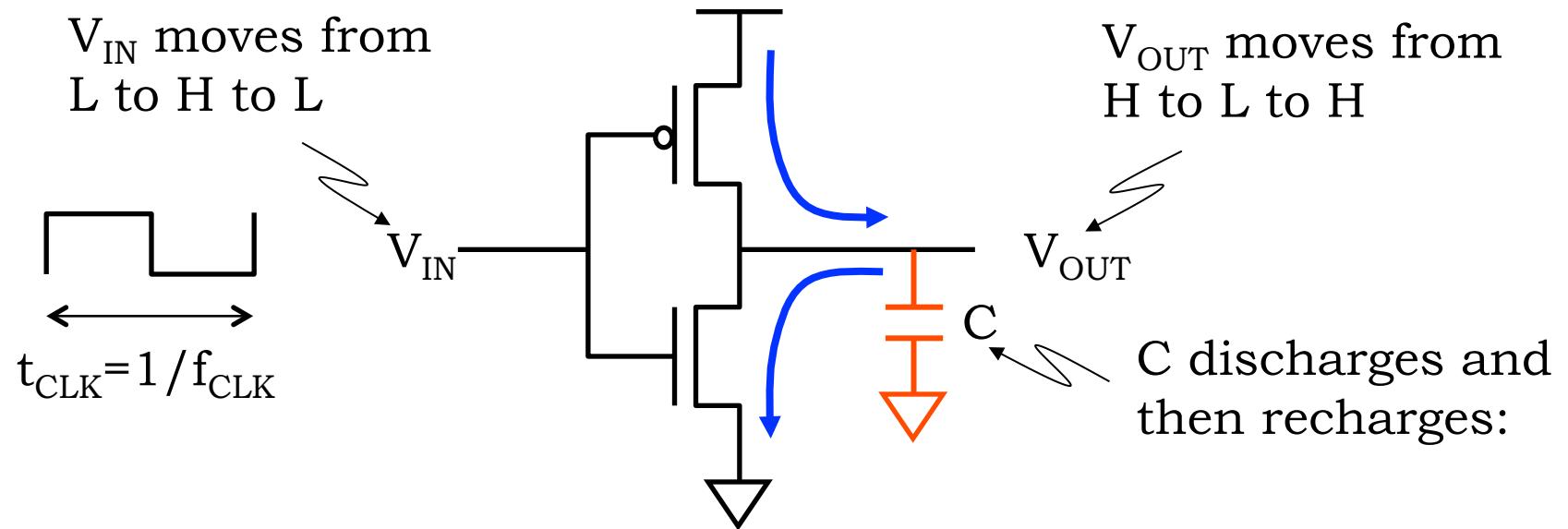
$$\begin{aligned} P_{NFET} &= f_{CLK} \int_0^{t_{CLK}/2} i_{NFET} V_{OUT} dt \\ &= f_{CLK} \int_0^{t_{CLK}/2} -C \frac{dV_{OUT}}{dt} V_{OUT} dt \\ &= f_{CLK} C \int_{V_{DD}}^0 -V_{OUT} dV_{OUT} \\ &= f_{CLK} C \frac{V_{DD}^2}{2} \end{aligned}$$



Power dissipated to recharge C:

$$\begin{aligned} P_{PFET} &= f_{CLK} \int_{t_{CLK}/2}^{t_{CLK}} i_{PFET} V_{OUT} dt \\ &= f_{CLK} \int_{t_{CLK}/2}^{t_{CLK}} C \frac{dV_{OUT}}{dt} V_{OUT} dt \\ &= f_{CLK} C \int_0^{V_{DD}} V_{OUT} dV_{OUT} \\ &= f_{CLK} C \frac{V_{DD}^2}{2} \end{aligned}$$

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 = frequency of charge/discharge

N = number of changing nodes/chip

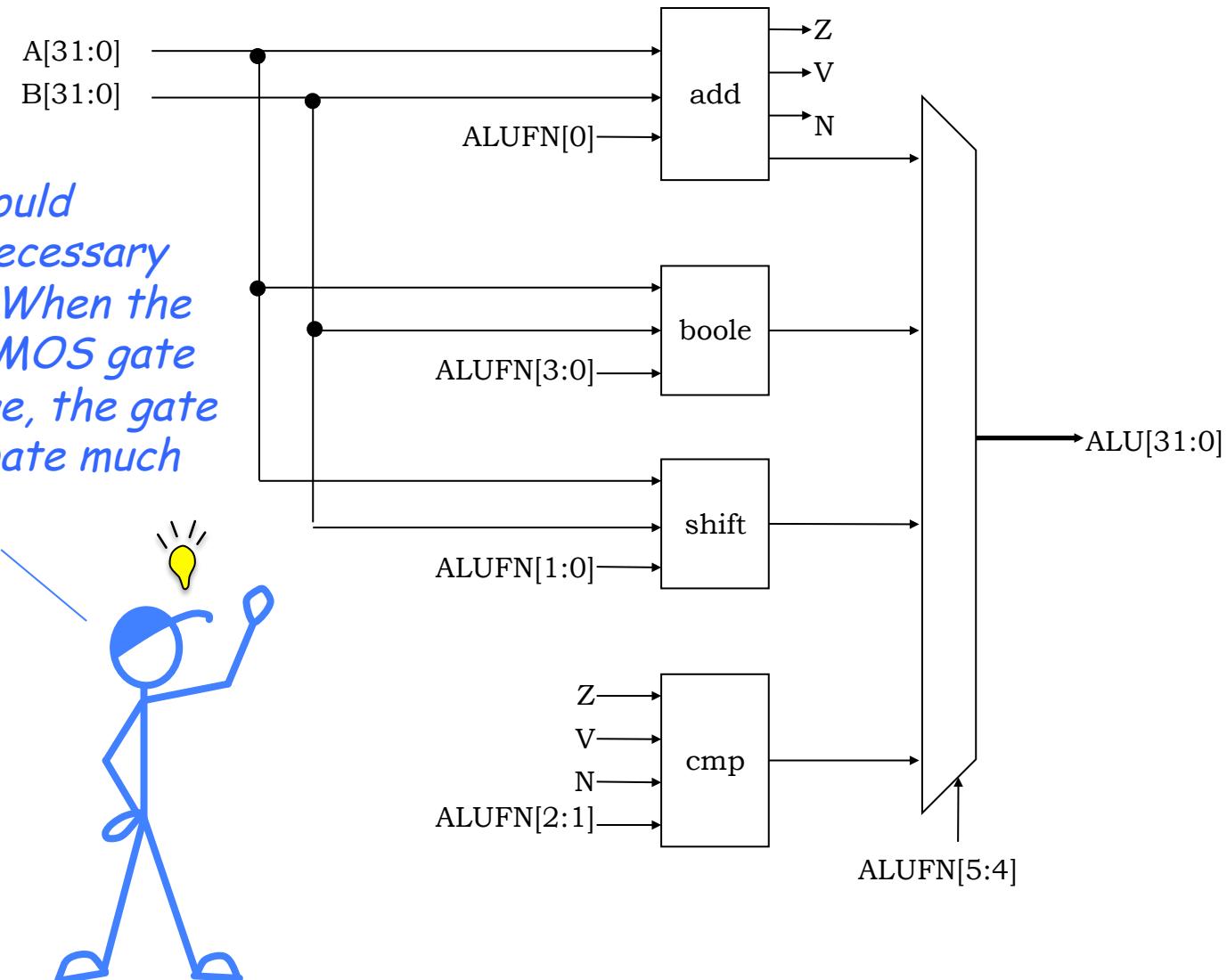
“Back of the envelope”: trends

$f \sim 1\text{GHz} = 1\text{e}9 \text{ cycles/sec}$	↑
$N \sim 1\text{e}8 \text{ changing nodes/cycle}$	↑
$C \sim 1\text{fF} = 1\text{e}-15 \text{ farads/node}$	↓
$V \sim 1\text{V}$	↔

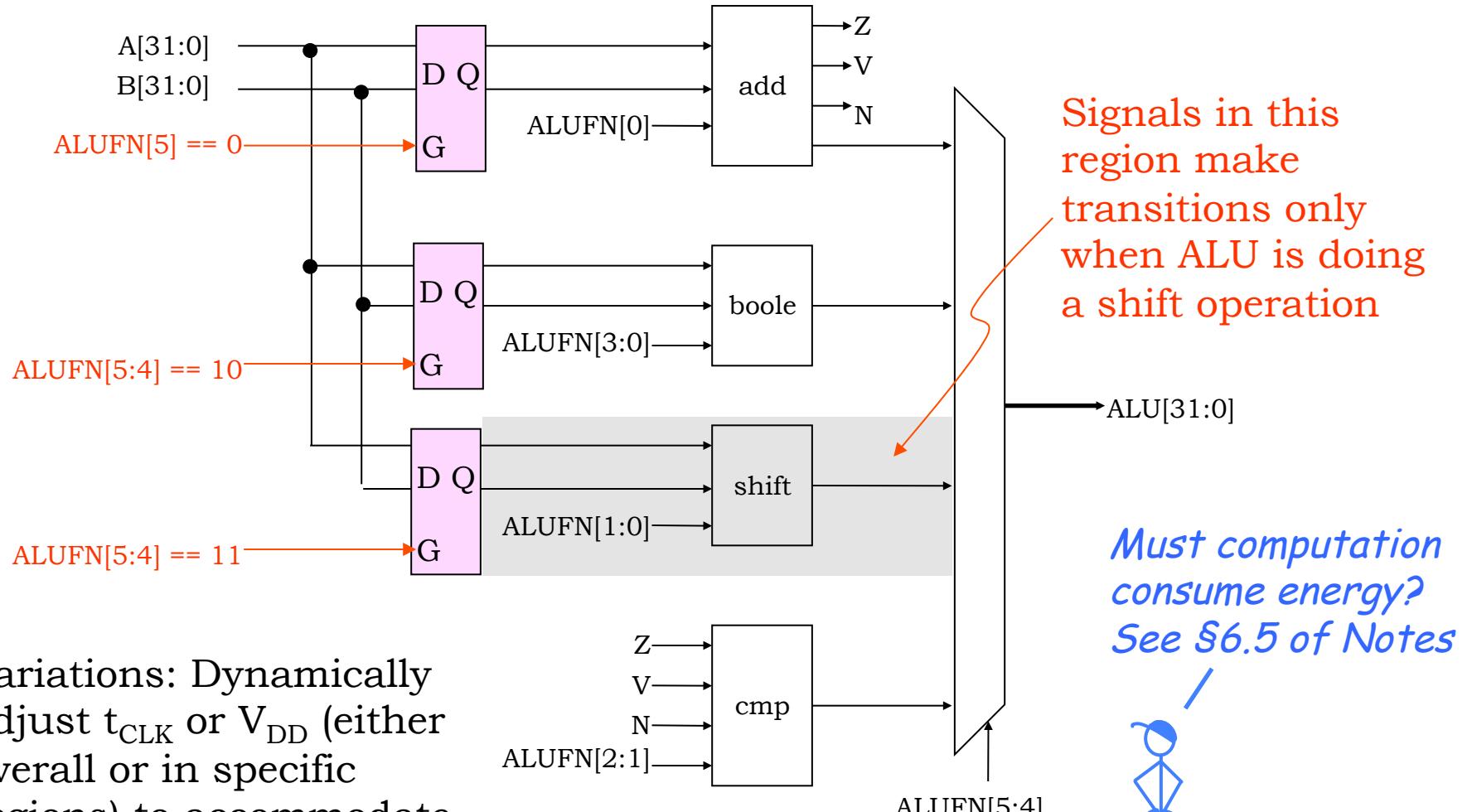
$\Rightarrow 100 \text{ Watts}$

How Can We Reduce Power?

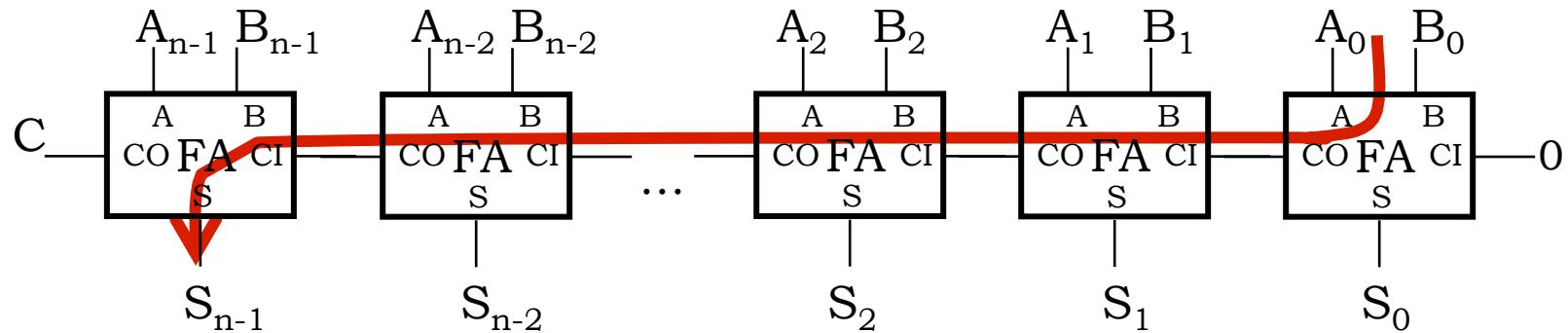
What if we could eliminate unnecessary transitions? When the output of a CMOS gate doesn't change, the gate doesn't dissipate much power!



Fewer Transitions → Lower Power



Improving Speed: Adder Example



Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001.

$$t_{PD} = (N-1) * (t_{PD,NAND3} + t_{PD,NAND2}) + t_{PD,XOR} \approx \Theta(N)$$

$\underbrace{\hspace{10em}}_{\text{CI to CO}}$ $\underbrace{\hspace{2em}}_{\text{CI}_{N-1} \text{ to } S_{N-1}}$

$\Theta(N)$ is read “order N” and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

Performance/Cost Analysis

“Order Of” notation:

" $g(n)$ is of order $f(n)$ " $g(n) = \Theta(f(n))$

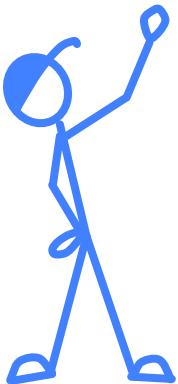
$g(n)=\Theta(f(n))$ if there exist $C_2 \geq C_1 > 0$
such that for all but finitely many
integral $n \geq 0$

$$C_1 \cdot f(n) \leq g(n) \leq C_2 \cdot f(n)$$



$\Theta(\dots)$ implies both
inequalities;

$O(\dots)$ implies only
the second.



$$g(n) = O(f(n))$$

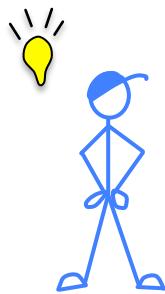
Example:

$$n^2 + 2n + 3 = \Theta(n^2)$$

since

$$n^2 < n^2 + 2n + 3 < 2n^2$$

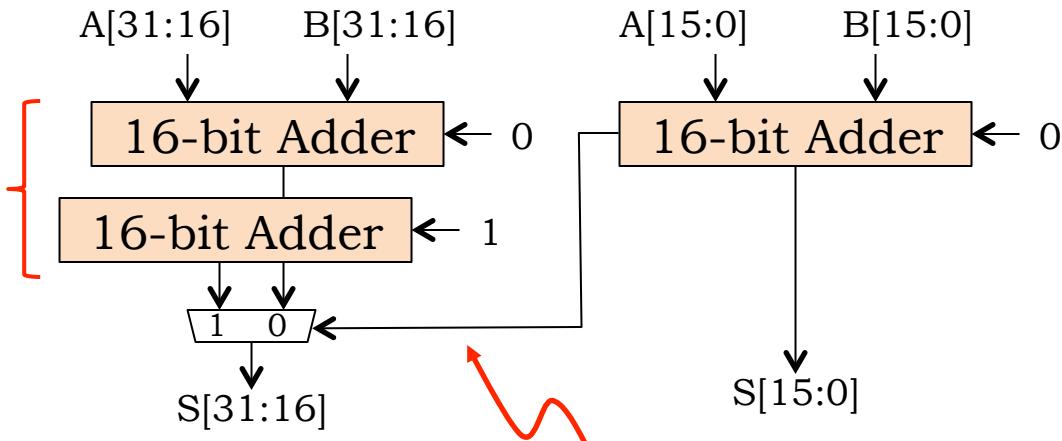
“almost always”



Carry Select Adders

Hmm. Can we get the high half of the adder working in parallel with the low half?

Two copies of the high half of the adder: one assumes a carry-in of "0", the other carry-in of "1".



Once the low half computes the actual value of the carry-in to the high half, use it select the correct version of the high-half addition.

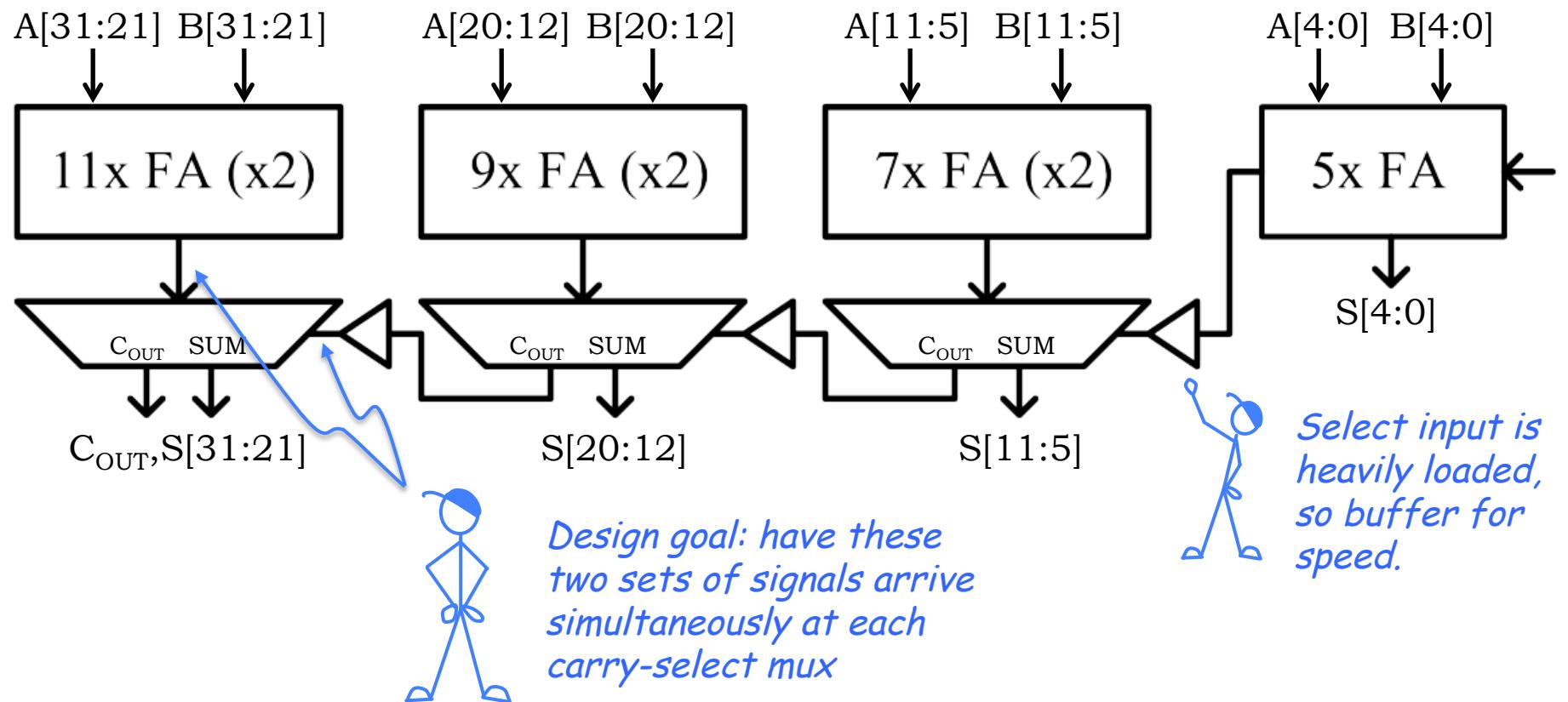
$$t_{PD} = 16*t_{PD,CI \rightarrow CO} + t_{PD,MUX2} \approx \text{half of } 32*t_{PD,CI \rightarrow CO}$$



Aha! Apply the same strategy to build 16-bit adders from 8-bit adders. And 8-bit adders from 4-bit adders, and so on. Resulting t_{PD} for N-bit adder is $\Theta(\log N)$.

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.



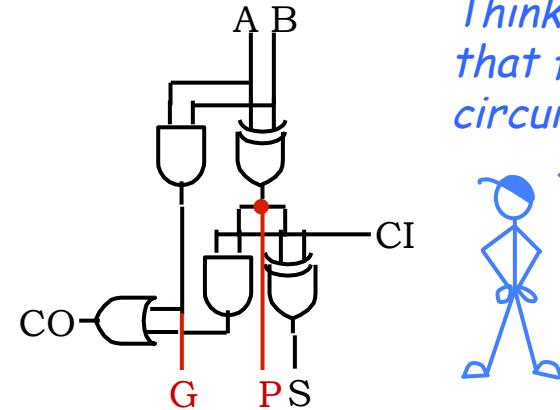
Wanted: Faster Carry Logic!

Let's see if we can improve the speed by rewriting the equations for C_{OUT} :

$$\begin{aligned} C_{OUT} &= AB + AC_{IN} + BC_{IN} \\ &= AB + (A + B)C_{IN} \\ &= \underset{\text{generate}}{G} + \underset{\text{propagate}}{P} C_{IN} \quad \text{where } G = AB \text{ and } P = A + B \end{aligned}$$

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} :

$$S = P \oplus C_{IN}$$



*CO logic using only
3 NAND2 gates!
Think I'll borrow
that for my FA
circuit!*

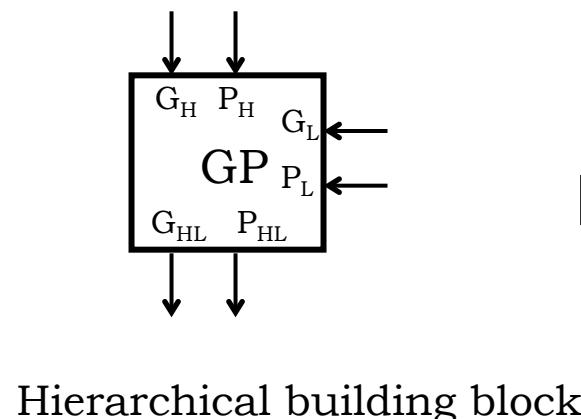
Carry Look-ahead Adders (CLA)

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:

$$G_{HL} = G_H + P_H G_L$$

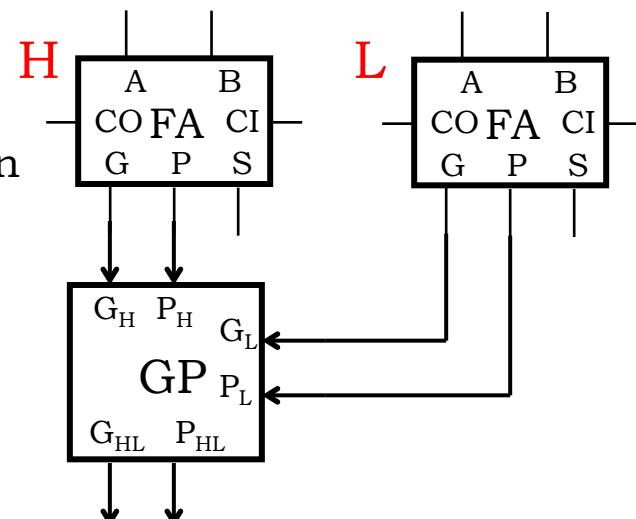
$$P_{HL} = P_H P_L$$

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.

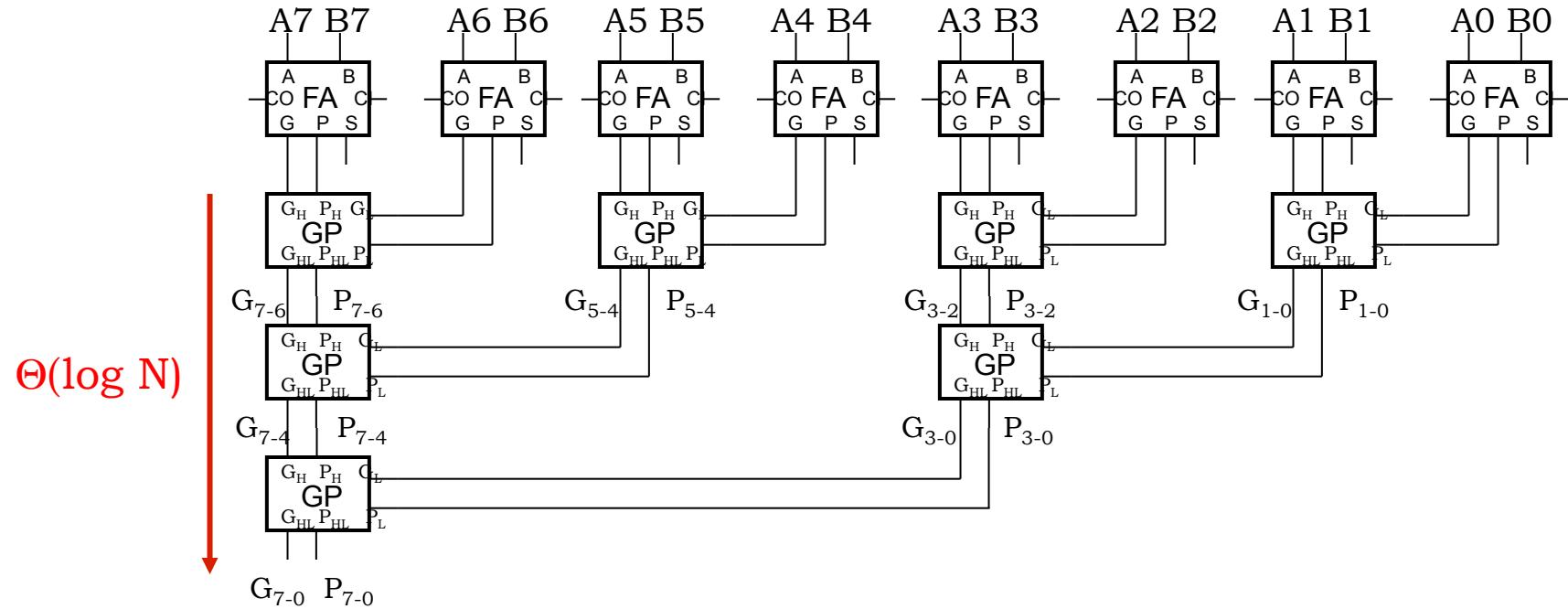


P/G generation

1st level of lookahead



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.

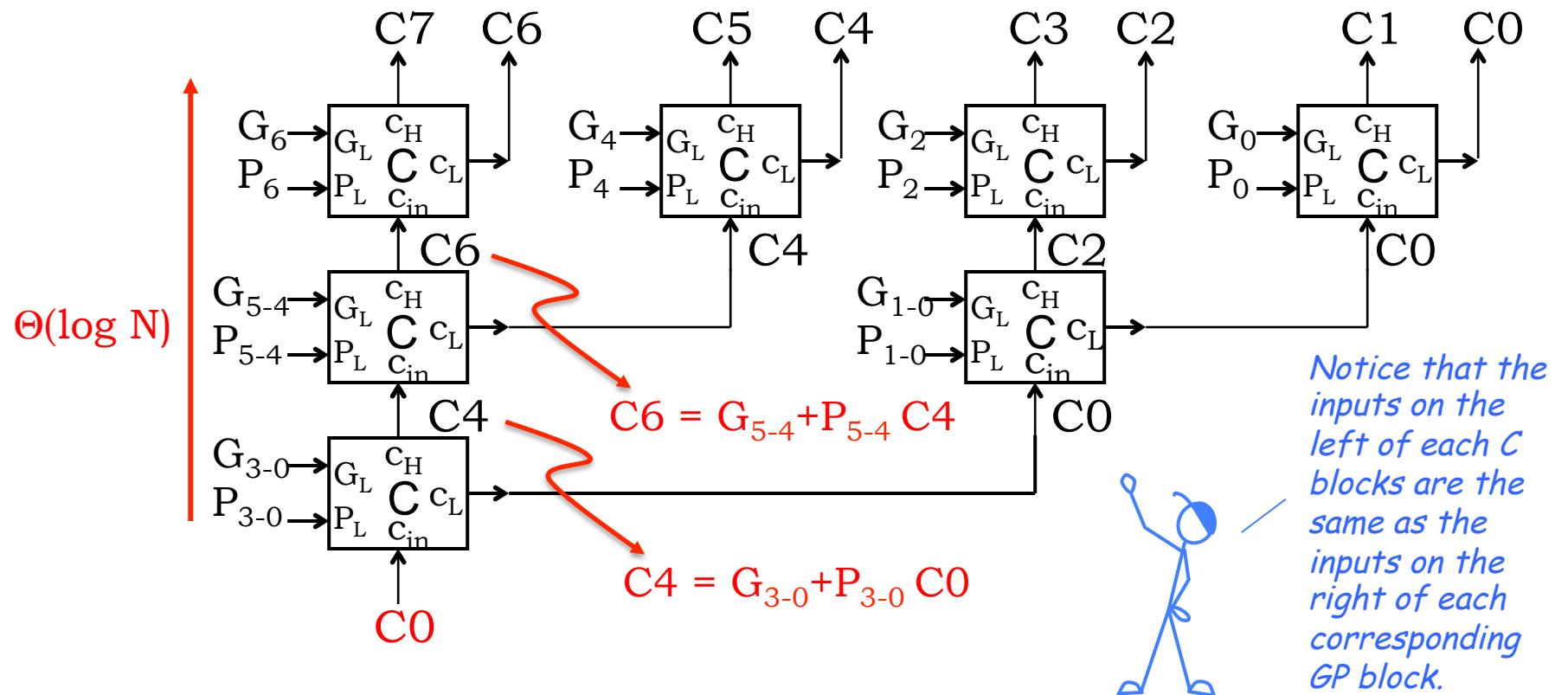
This will let us quickly compute the carry-ins for each FA!

8-bit CLA (carry generation)

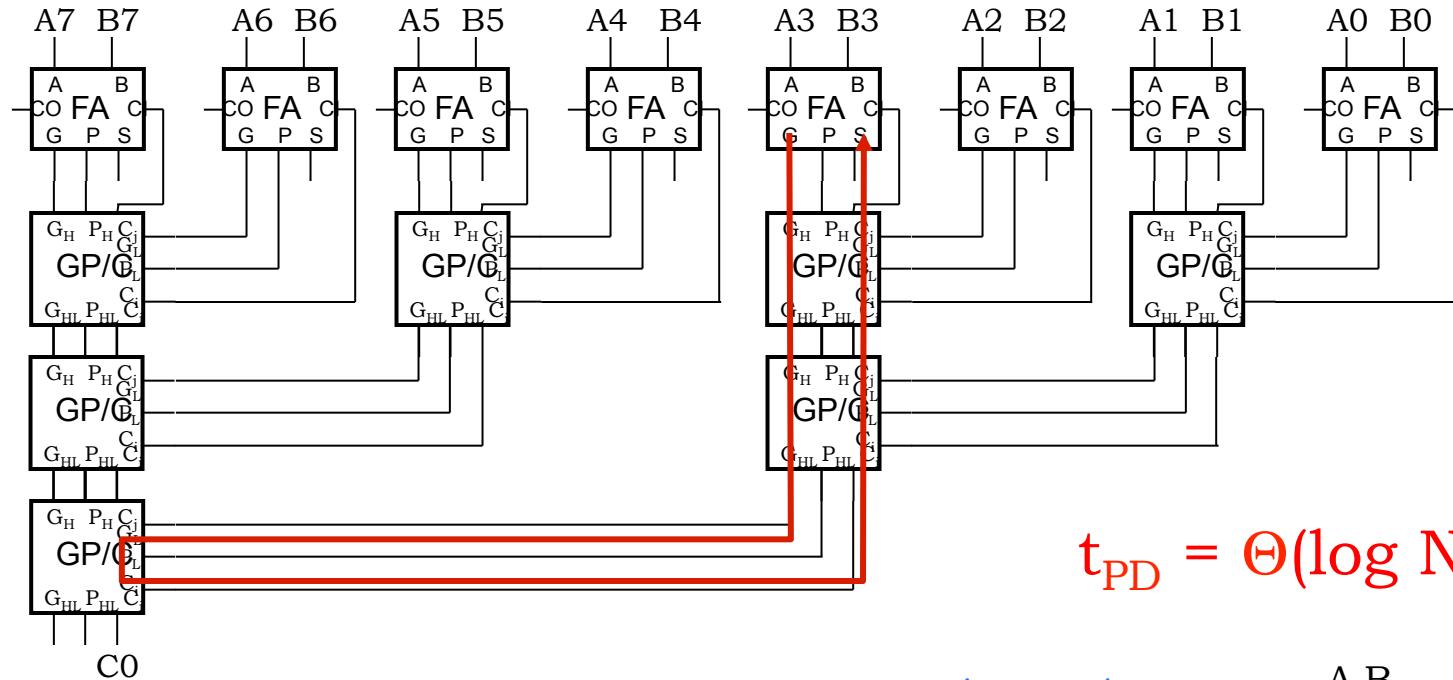
Now, given a the value of the carry-in of the least-significant bit, we can generate the carries for every adder.

$$C_H = G_L + P_L C_{in}$$

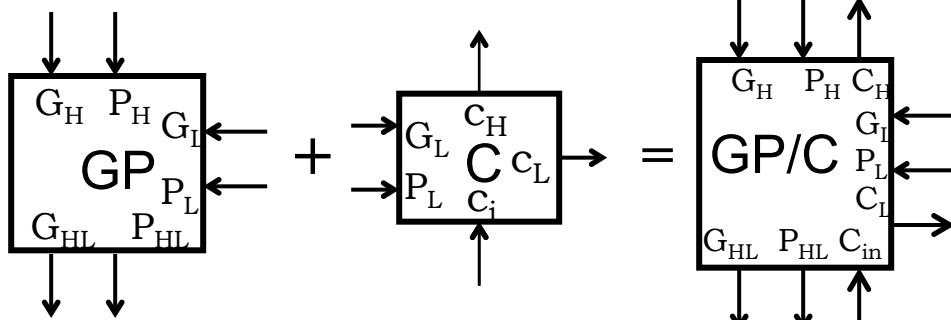
$$C_L = C_{in}$$



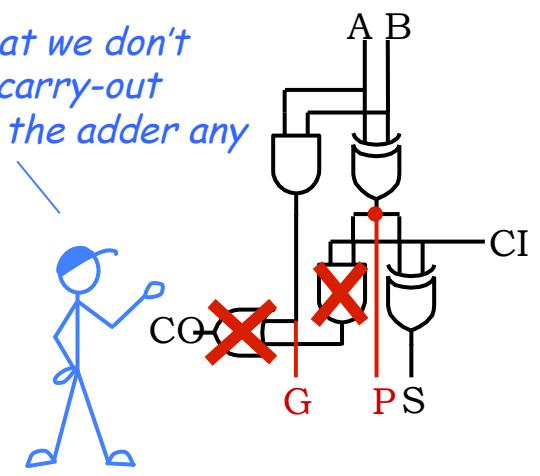
8-bit CLA (complete)



$$t_{PD} = \Theta(\log N)$$



Notice that we don't need the carry-out output of the adder any more.



To learn more, look up Kogge-Stone adders on Wikipedia.

Binary Multiplication*

The “Binary”
Multiplication
Table

Hey, that
looks like
an AND
gate

*	0	1
0	0	0
1	0	1

* Actually unsigned binary multiplication

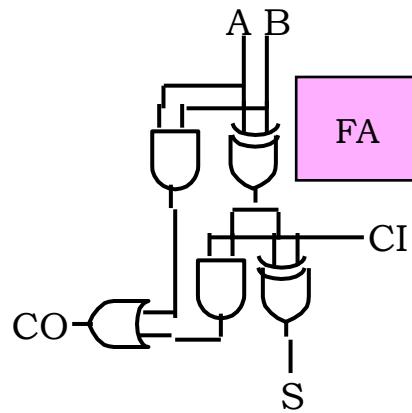
$$\begin{array}{r} A_3 \quad A_2 \quad A_1 \quad A_0 \\ \times \quad B_3 \quad B_2 \quad B_1 \quad B_0 \\ \hline \end{array}$$

AB_i called a “partial product” \longrightarrow

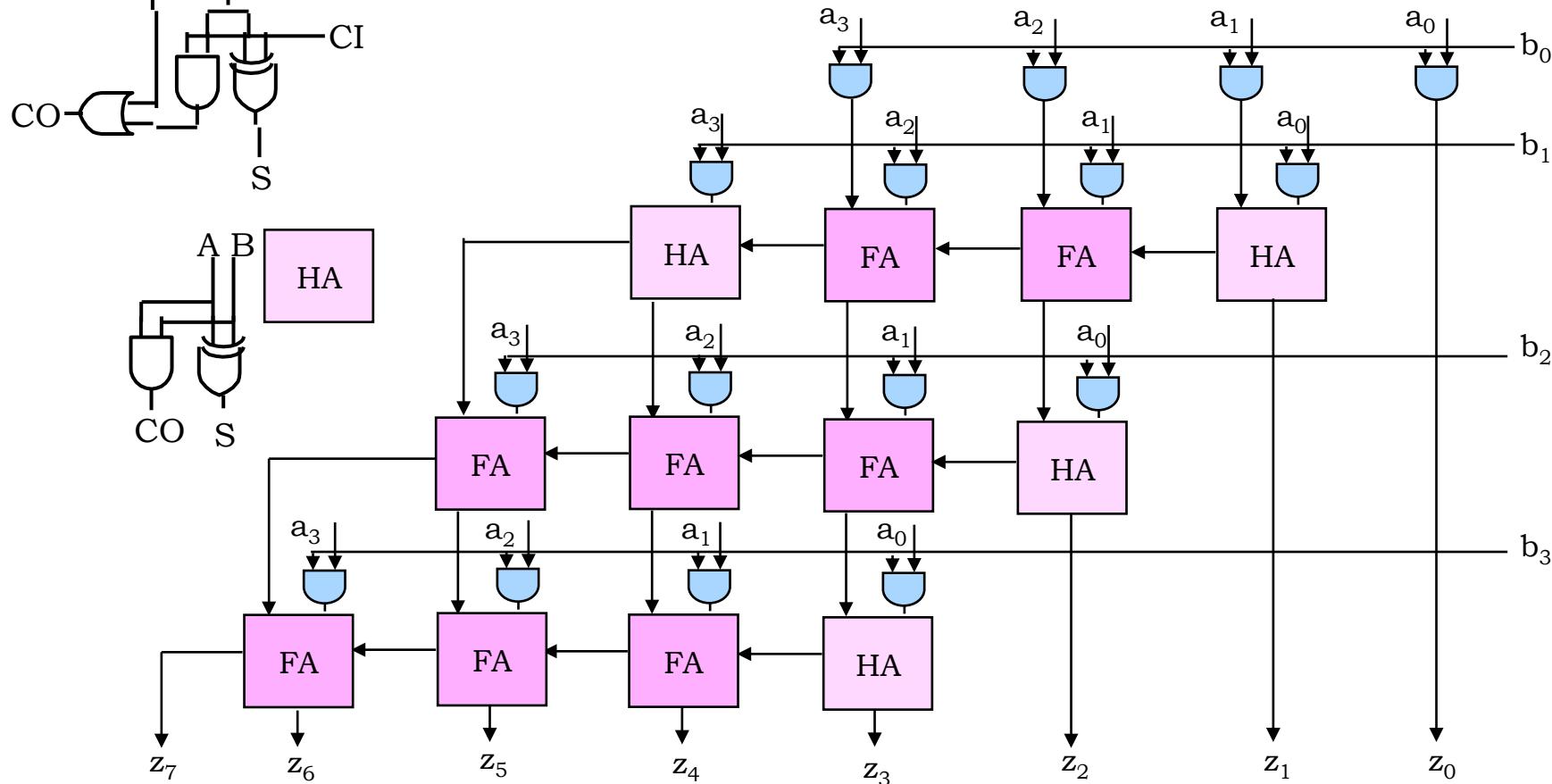
$$\begin{array}{cccc} A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\ A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\ A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\ + \quad A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\ \hline \end{array}$$

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



Combinational Multiplier



Latency = $\Theta(N)$
 Throughput = $\Theta(1/N)$
 Hardware = $\Theta(N^2)$

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

$$\begin{array}{r}
 & \text{x3} & \text{x2} & \text{x1} & \text{x0} \\
 * & \text{y3} & \text{y2} & \text{y1} & \text{y0} \\
 \hline
 \end{array}$$

$$\begin{array}{r}
 \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x2y0} & \text{x1y0} & \text{x0y0} \\
 + \text{x3y1} & \text{x3y1} & \text{x3y1} & \text{x3y1} & \text{x2y1} & \text{x2y1} & \text{x1y1} & \text{x0y1} \\
 + \text{x3y2} & \text{x3y2} & \text{x3y2} & \text{x2y2} & \text{x1y2} & \text{x0y2} \\
 - \text{x3y3} & \text{x3y3} & \text{x2y3} & \text{x1y3} & \text{x0y3} \\
 \hline
 \end{array}$$

$$\begin{array}{cccccccc}
 \text{z7} & \text{z6} & \text{z5} & \text{z4} & \text{z3} & \text{z2} & \text{z1} & \text{z0} \\
 \hline
 \end{array}$$

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).

$$\begin{array}{r}
 \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x2y0} & \text{x1y0} & \text{x0y0} \\
 + & & & & & \text{1} \\
 + \text{x3y1} & \text{x3y1} & \text{x3y1} & \text{x3y1} & \text{x2y1} & \text{x1y1} & \text{x0y1} \\
 + & & & & \text{1} \\
 + \text{x3y2} & \text{x3y2} & \text{x3y2} & \text{x2y2} & \text{x1y2} & \text{x0y2} \\
 + & & & & \text{1} \\
 + \text{x3y3} & \text{x3y3} & \text{x2y3} & \text{x1y3} & \text{x0y3} & \left. \right\} -B = \sim B + 1 \\
 + & & & & \text{1} \\
 - & & \text{1} & \text{1} & \text{1} & \text{1} \\
 \end{array}$$

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

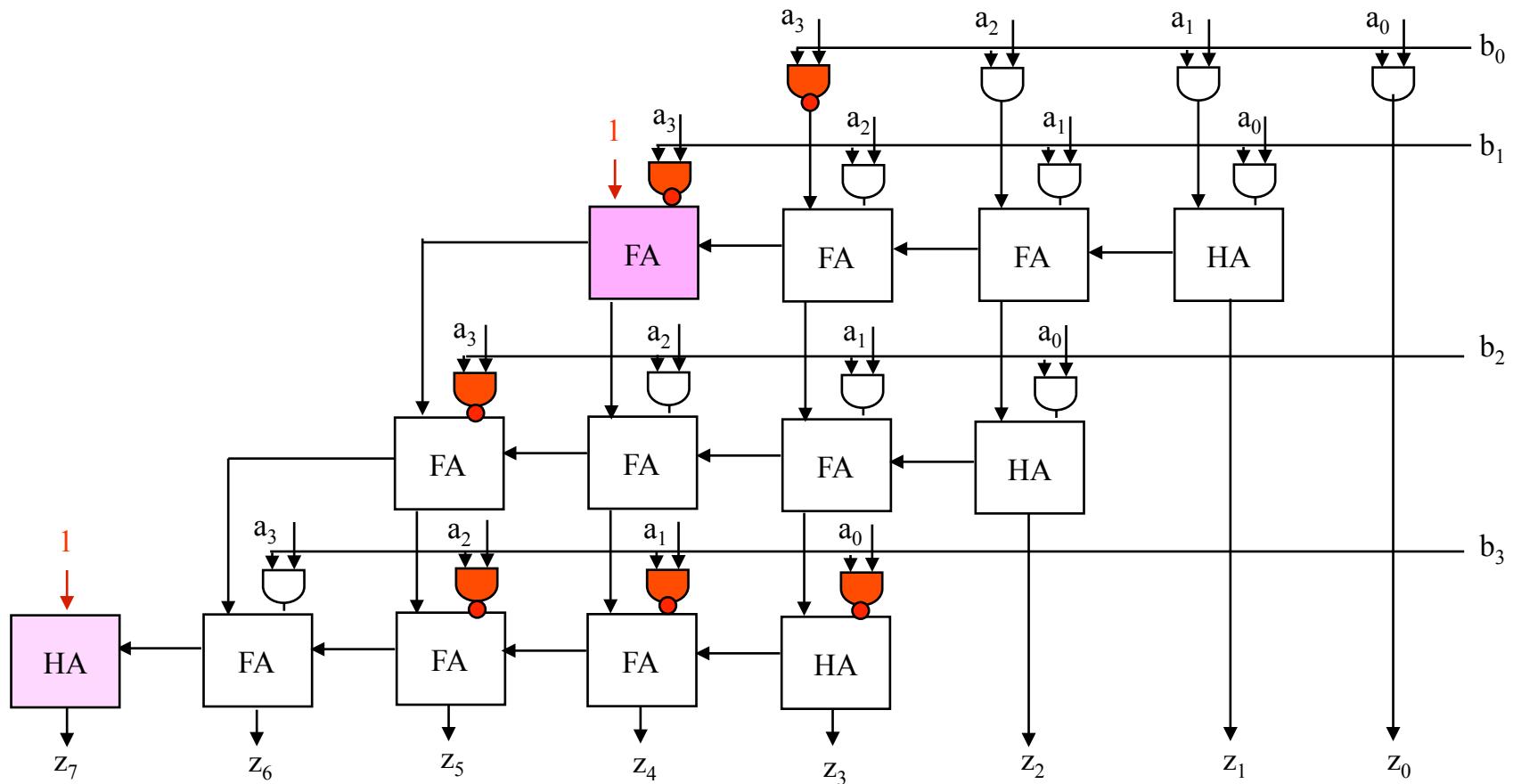
$$\begin{array}{r}
 & & & \text{x3y0} & \text{x2y0} & \text{x1y0} & \text{x0y0} \\
 & & & \text{x3y1} & \text{x2y1} & \text{x1y1} & \text{x0y1} \\
 & & & \text{x3y2} & \text{x2y2} & \text{x1y2} & \text{x0y2} \\
 & & & \text{x3y3} & \text{x2y3} & \text{x1y3} & \text{x0y3} \\
 & & & & & & 1 \\
 & & & 1 & 1 & 1 & 1 \\
 \end{array}$$

Step 4: finish computing the constants...

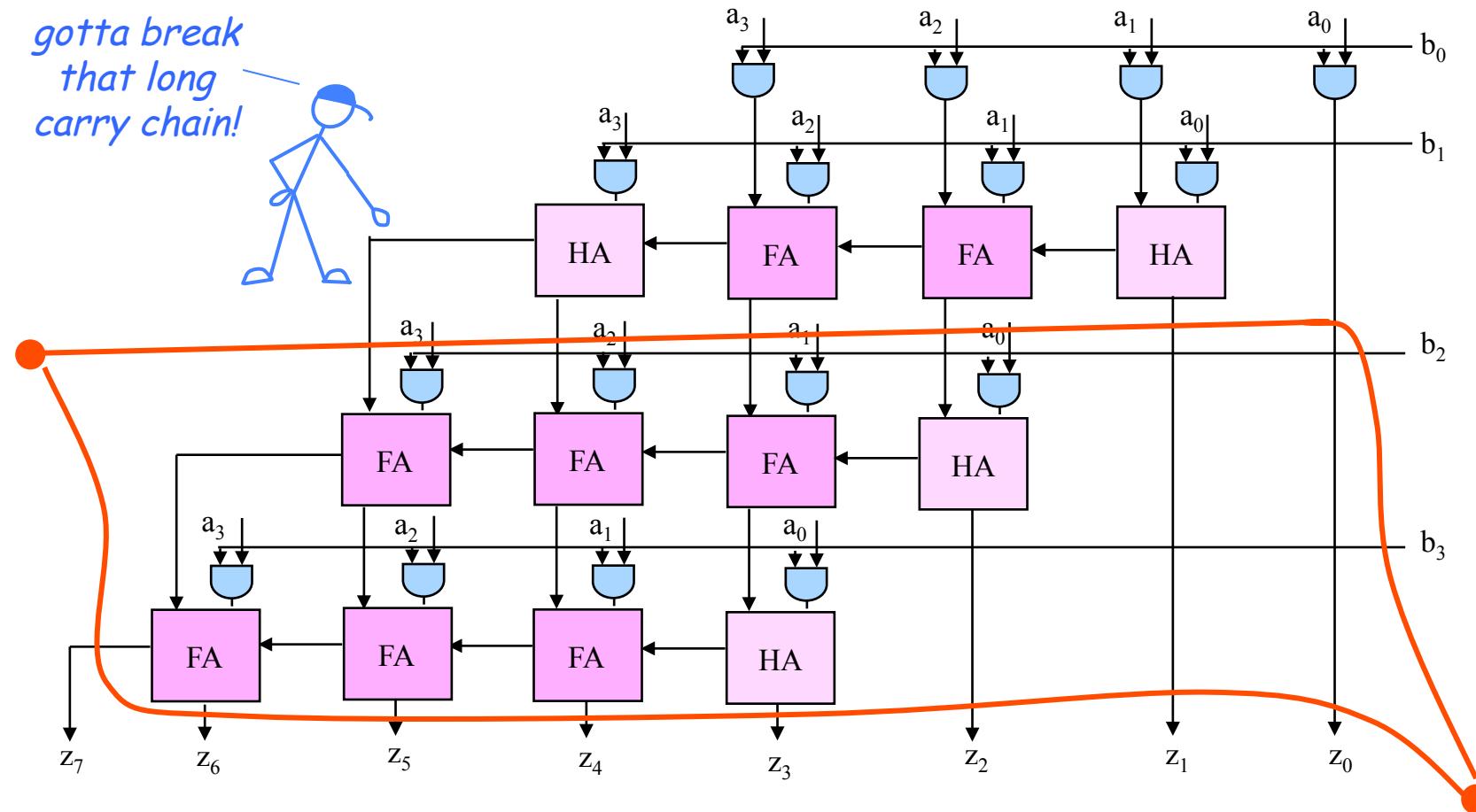
$$\begin{array}{r}
 & & & \text{x3y0} & \text{x2y0} & \text{x1y0} & \text{x0y0} \\
 & & & \text{x3y1} & \text{x2y1} & \text{x1y1} & \text{x0y1} \\
 & & & \text{x3y2} & \text{x2y2} & \text{x1y2} & \text{x0y2} \\
 & & & \text{x3y3} & \text{x2y3} & \text{x1y3} & \text{x0y3} \\
 & & & 1 & & 1 & \\
 \end{array}$$

Result: multiplying 2's complement operands takes just about same amount of hardware as multiplying unsigned operands!

2's Complement Multiplier



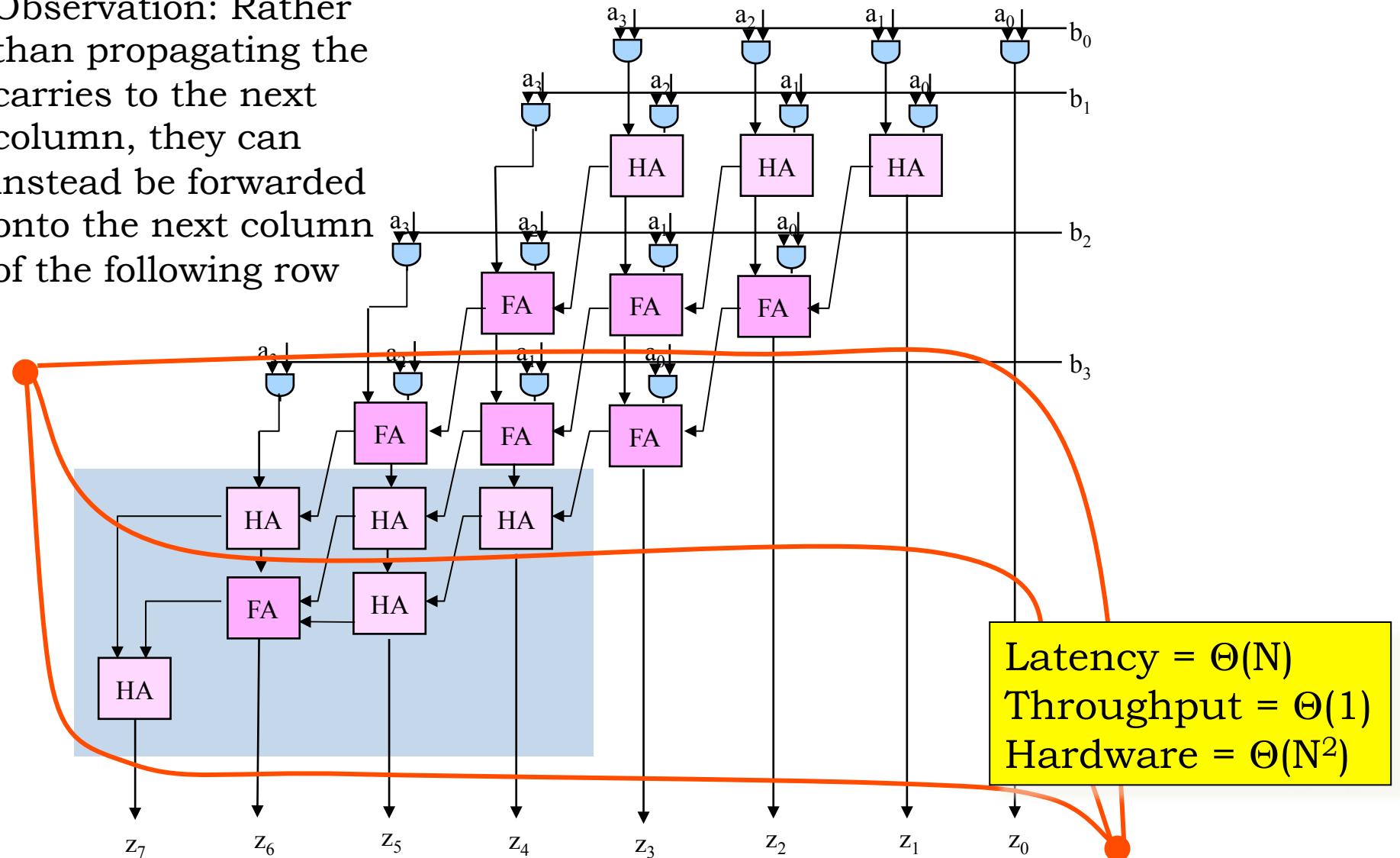
Increase Throughput With Pipelining



Before pipelining: Throughput = $\sim 1/(2N) = \Theta(1/N)$
After pipelining: Throughput = $\sim 1/N = \Theta(1/N)$

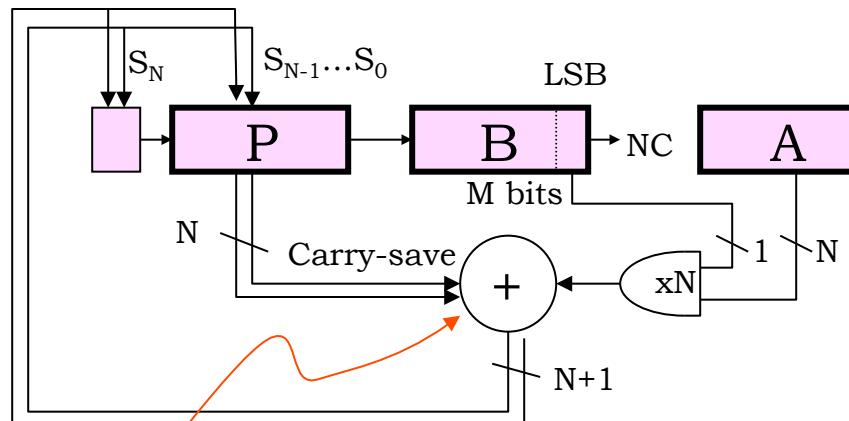
“Carry-save” Pipelined Multiplier

Observation: Rather than propagating the carries to the next column, they can instead be forwarded onto the next column of the following row



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:



$T_{PD} = \Theta(1)$ for carry-save (see previous slide),
but adds $\Theta(N)$ cycles & $\Theta(N)$ hardware

Init: $P \leftarrow 0$, load A&B
Repeat M times {
 $P \leftarrow P + (B_{LSB} == 1 ? A : 0)$
 shift S_N, P, B right one bit
}
Done: $(N+M)$ -bit result in P,B

Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Hardware = $\Theta(N)$

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.

9. Programmable Machines

6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2015 MIT EECS

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!

Example: Factorial

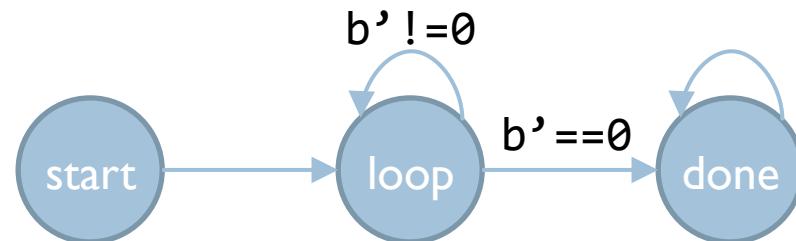
factorial(N) = N! = N*(N-1)*...*1

C:

```
int a = 1;  
int b = N;  
do {  
    a = a * b;  
    b = b - 1;  
} while (b != 0)
```

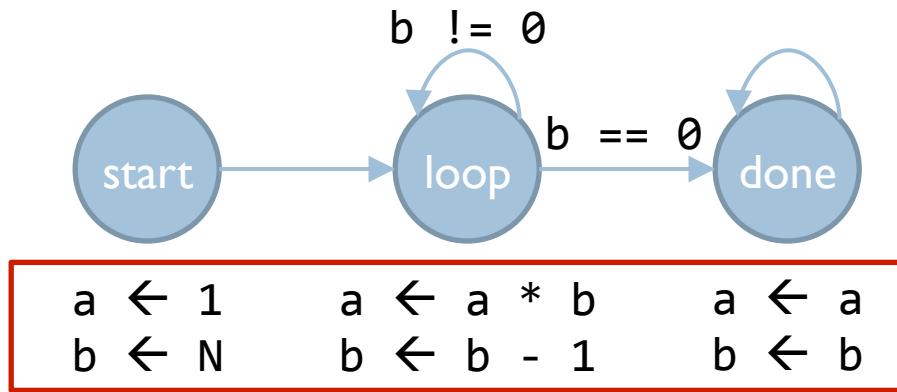
start: a \leftarrow 1, b \leftarrow 5
loop: a \leftarrow 5, b \leftarrow 4
loop: a \leftarrow 20, b \leftarrow 3
loop: a \leftarrow 60, b \leftarrow 2
loop: a \leftarrow 120, b \leftarrow 1
loop: a \leftarrow 120, b \leftarrow 0
done:

High-level FSM:

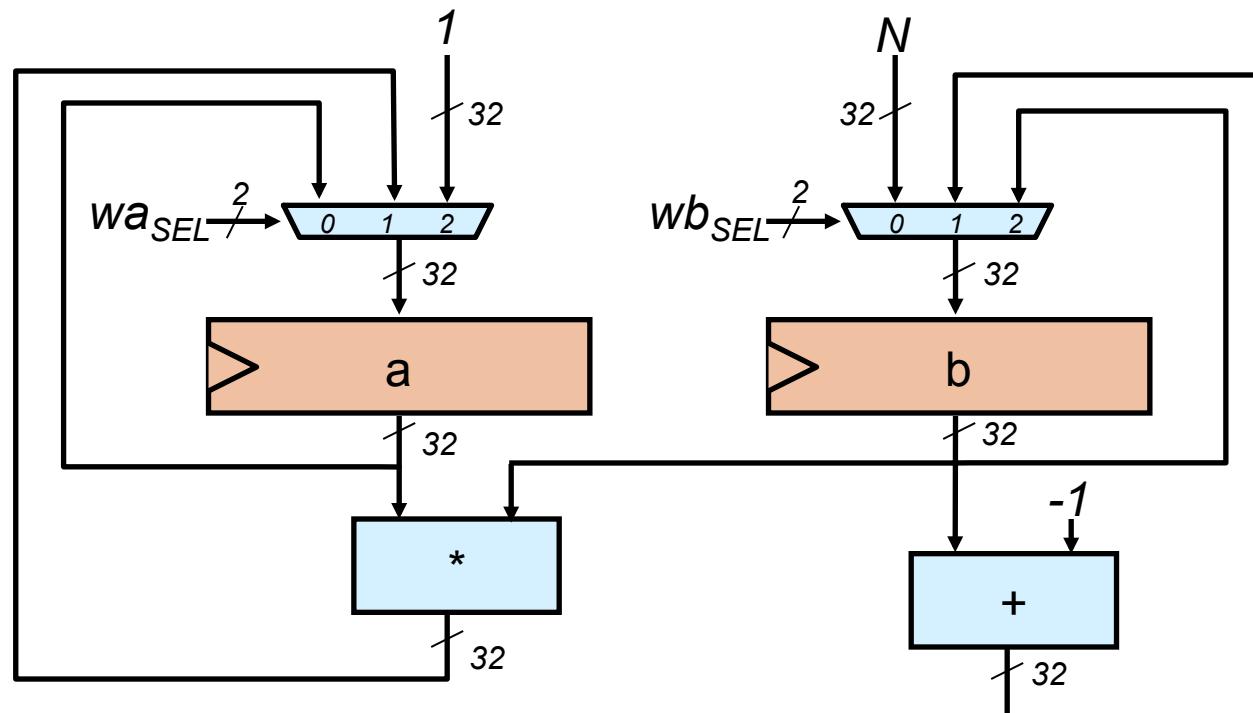


- Helpful to translate into hardware
- **D-registers** (a, b)
- 2-bits of state (start, loop, done)
- Boolean transitions ($b' == 0$, $b' != 0$)
- **Register assignments** in states
(e.g., $a \leftarrow a * b$)

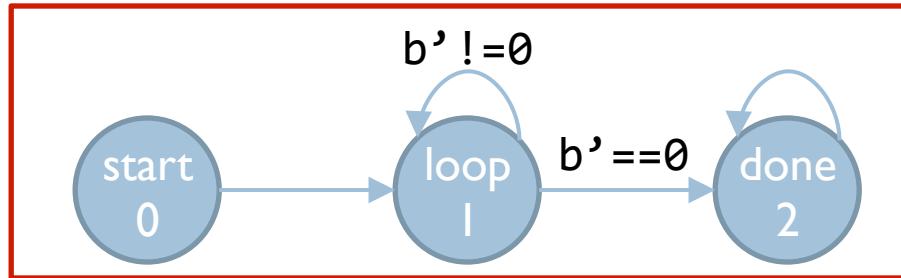
Datapath for Factorial



- Draw registers
- Draw combinational circuit for each assignment
- Connect to input muxes



Control FSM for Factorial

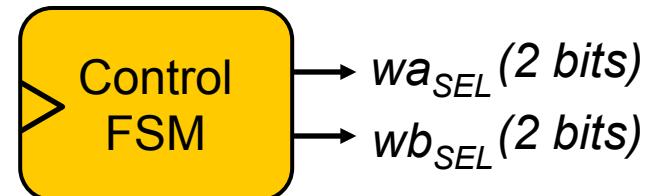
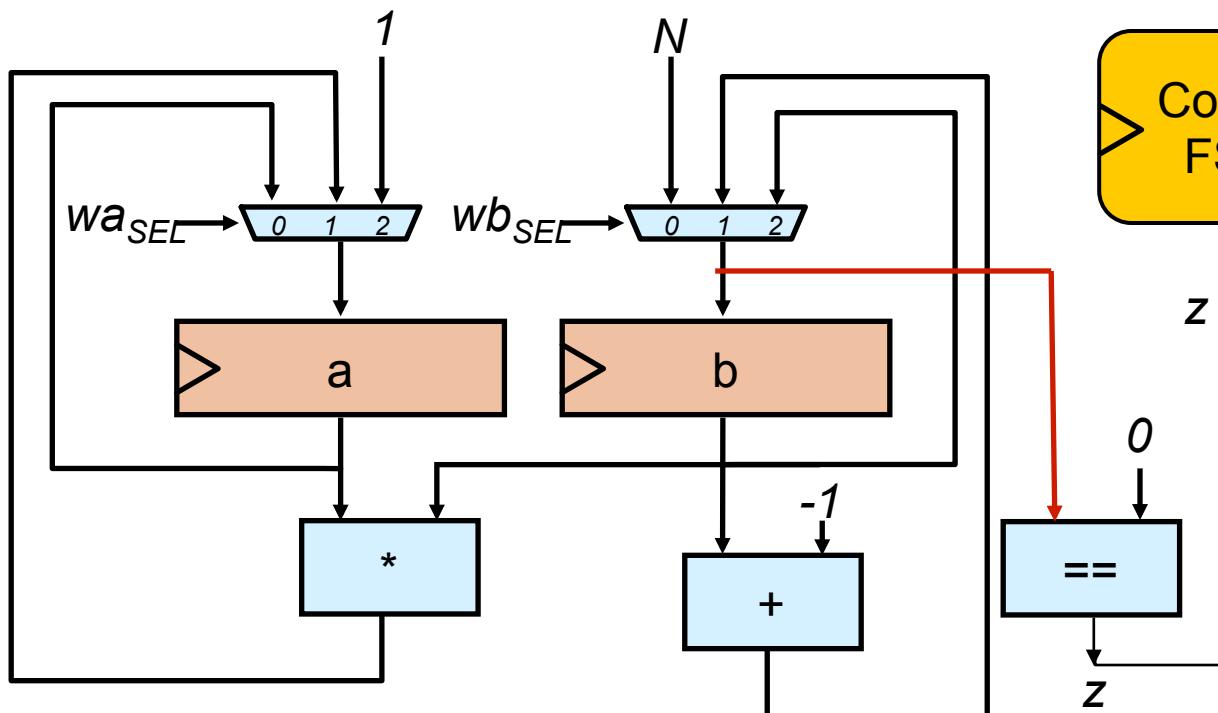


$a \leftarrow 1$
 $b \leftarrow N$

$a \leftarrow a * b$
 $b \leftarrow b - 1$

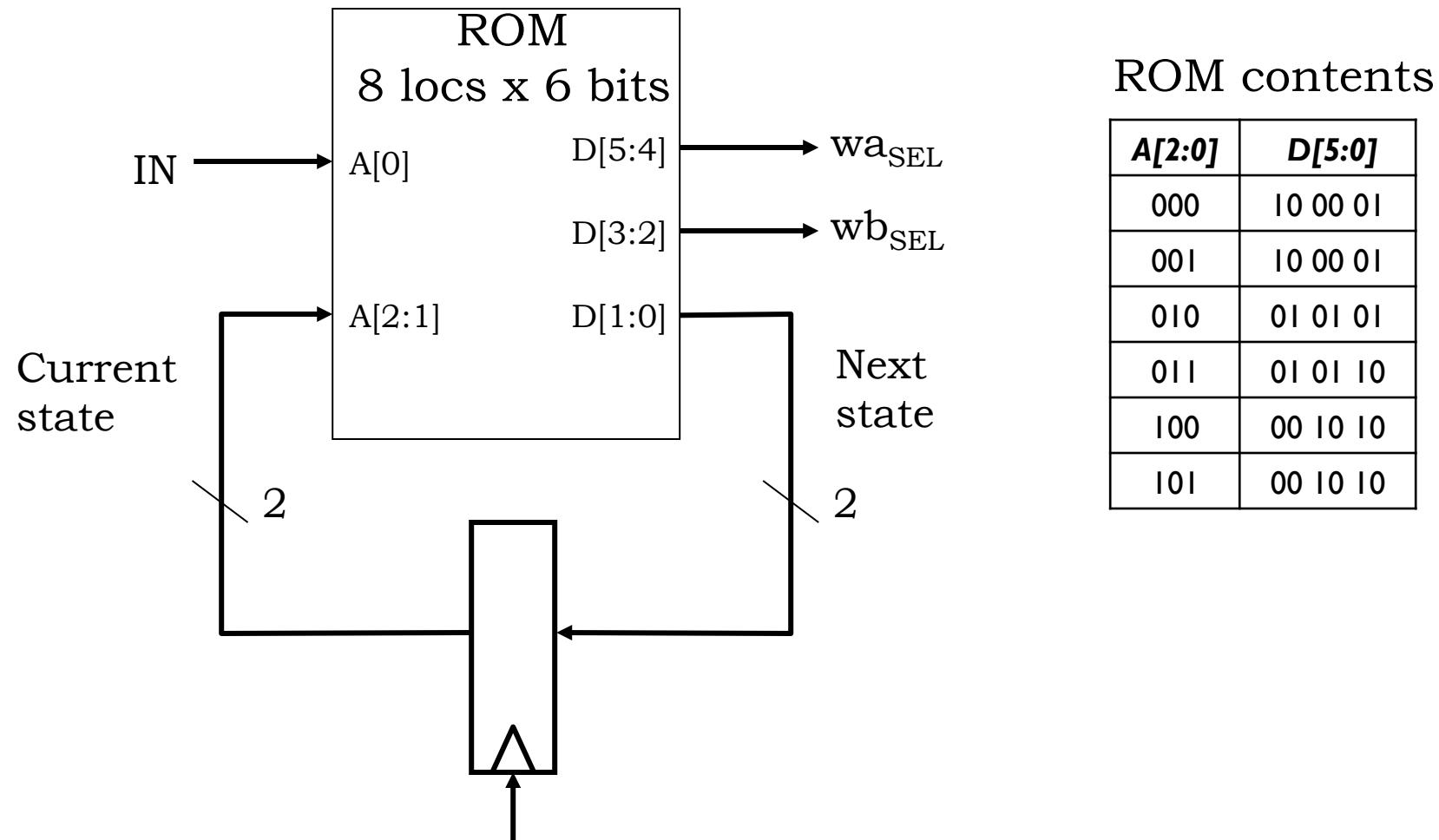
$a \leftarrow a$
 $b \leftarrow b$

- Draw combinational logic for transition conditions
- Implement control FSM:
 - States: High-level FSM states
 - Inputs: Transition logic outputs
 - Outputs: Mux select signals



S	Z	wa _{SEL}	wb _{SEL}	S'
00	0	10	00	01
00	1	10	00	01
01	0	01	01	01
01	1	01	01	10
10	0	00	10	10
10	1	00	10	10

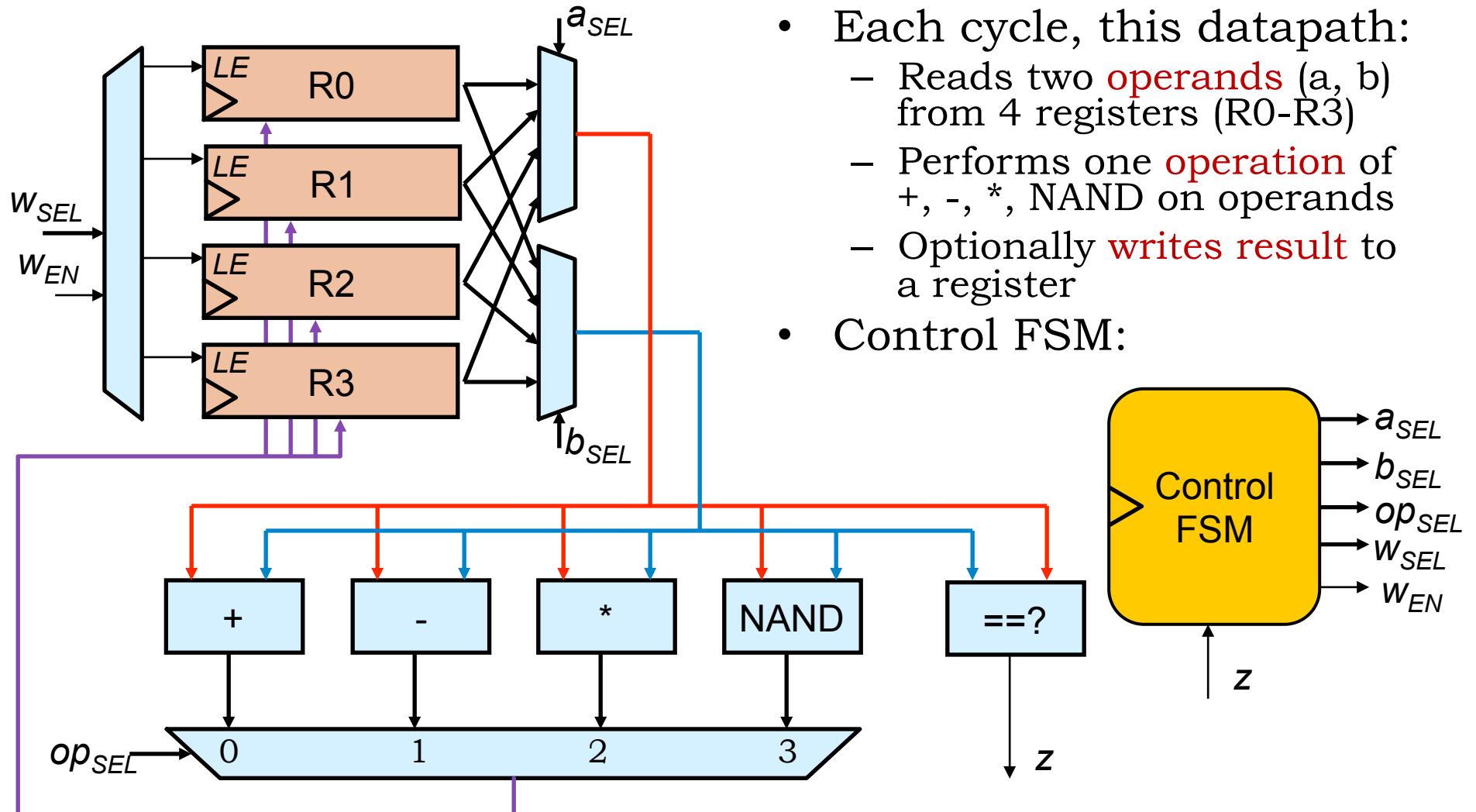
Control FSM Hardware



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

A Simple Programmable Datapath



A Control FSM for Factorial

- Assume initial register contents:

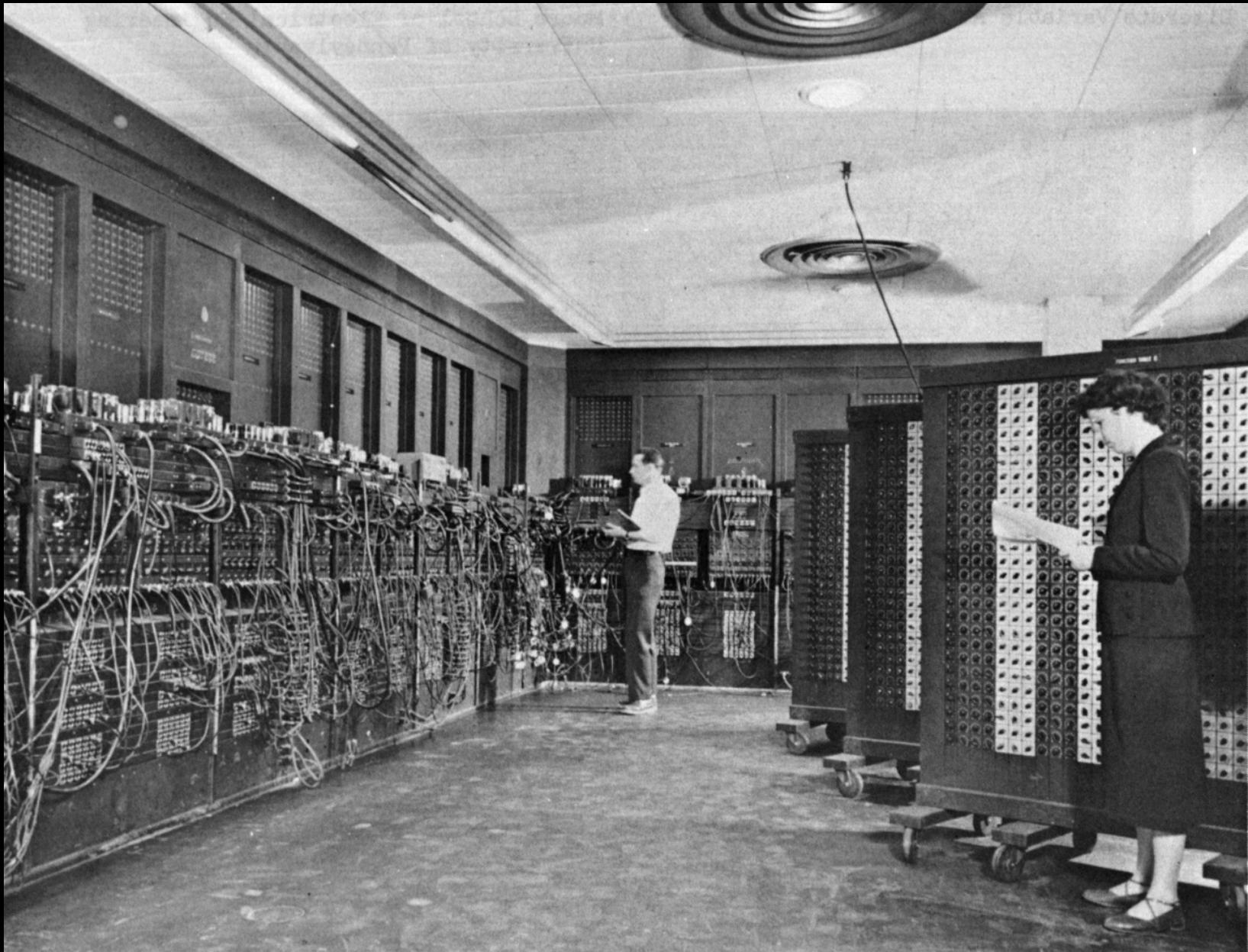
R0 value = 1
R1 value = N
R2 value = -1
R3 value = 0
- Control FSM:

```
graph LR; start(( )) --> loopMul((loop mul)); loopMul -- "z == 0" --> loopSub((loop sub)); loopSub -- "z == 0" --> loopBeq((loop beq)); loopBeq -- "z == 1" --> done((done)); loopBeq -- "z == 1" --> loopMul;
```

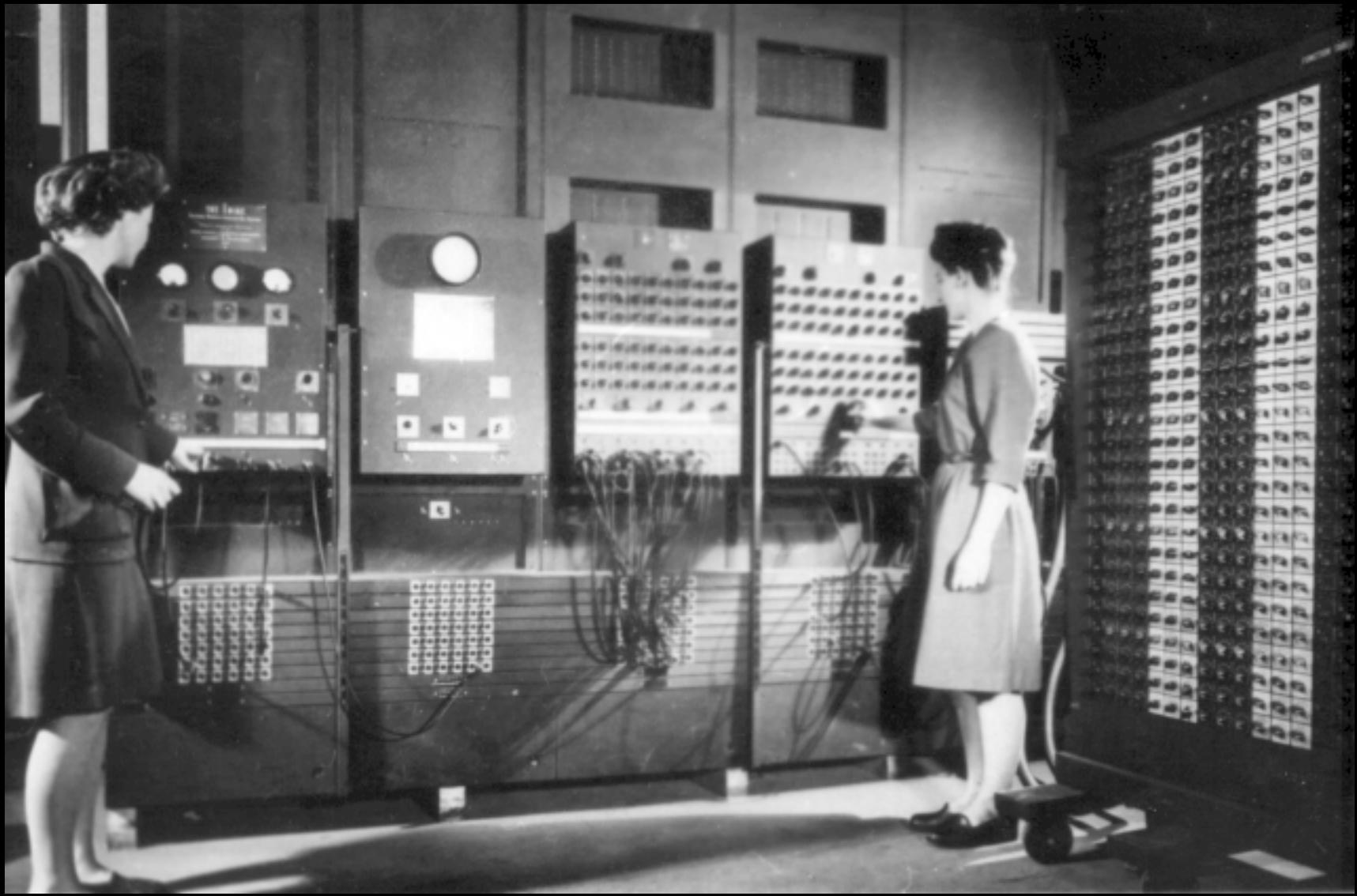
State	Condition	Next State	Register Values	Operations
loop mul	Initial	loop sub	asel = 0 bsel = 1 opsel = 2 (*) wen = 1 wsel = 0	$R0 \leftarrow R0 * R1$
loop sub	z == 0	loop beq	asel = 1 bsel = 2 opsel = 0 (+) wen = 1 wsel = 1	$R1 \leftarrow R1 + R2$
loop beq	z == 0	done	asel = 1 bsel = 3 opsel = X wen = 0 wsel = X	
done	z == 1	done	asel = 1 bsel = 3 opsel = X wen = 0 wsel = X	N! in R0

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



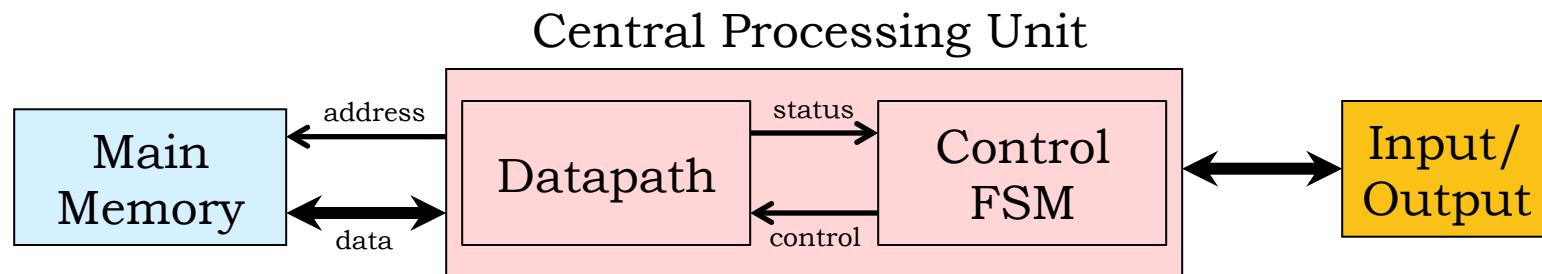
"Eniac" by Unknown - U.S. Army Photo.



U.S. Army Photo.

The von Neumann Model

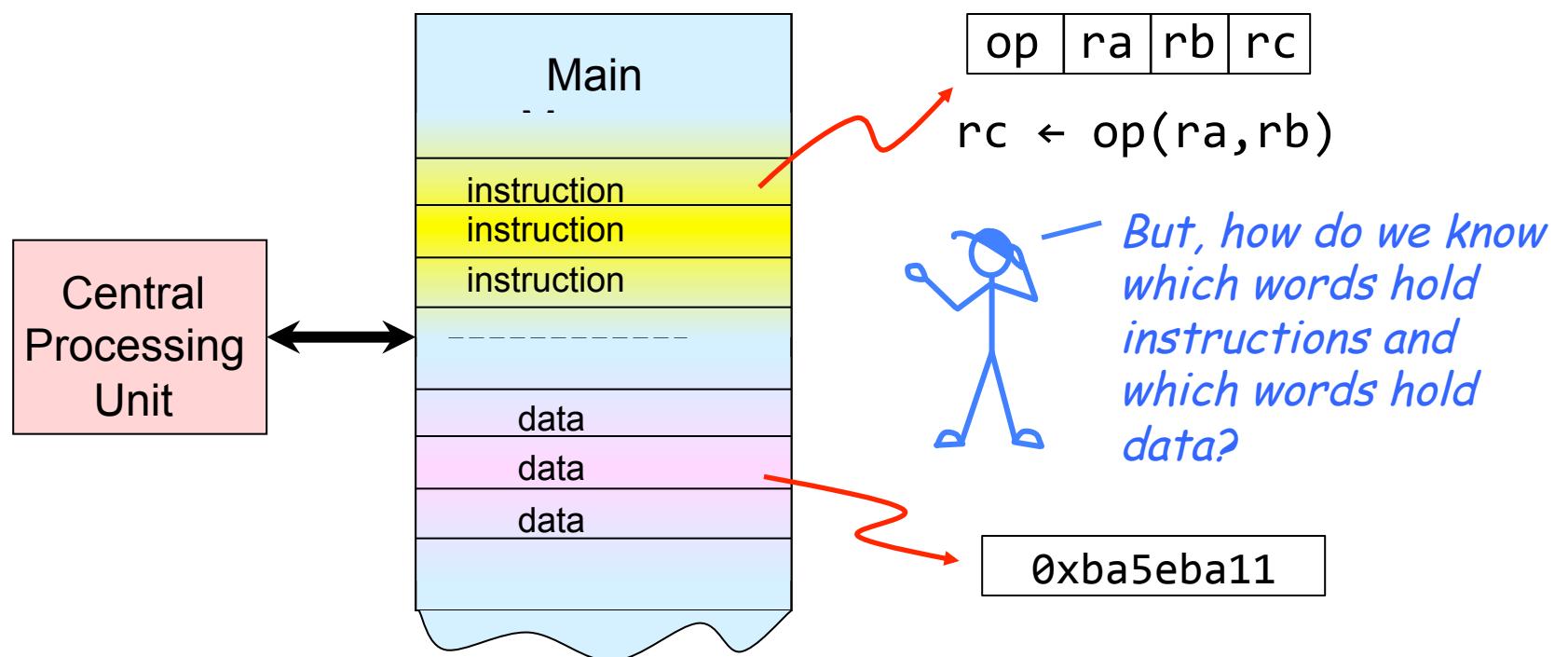
- Many approaches to build a general-purpose computer. Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



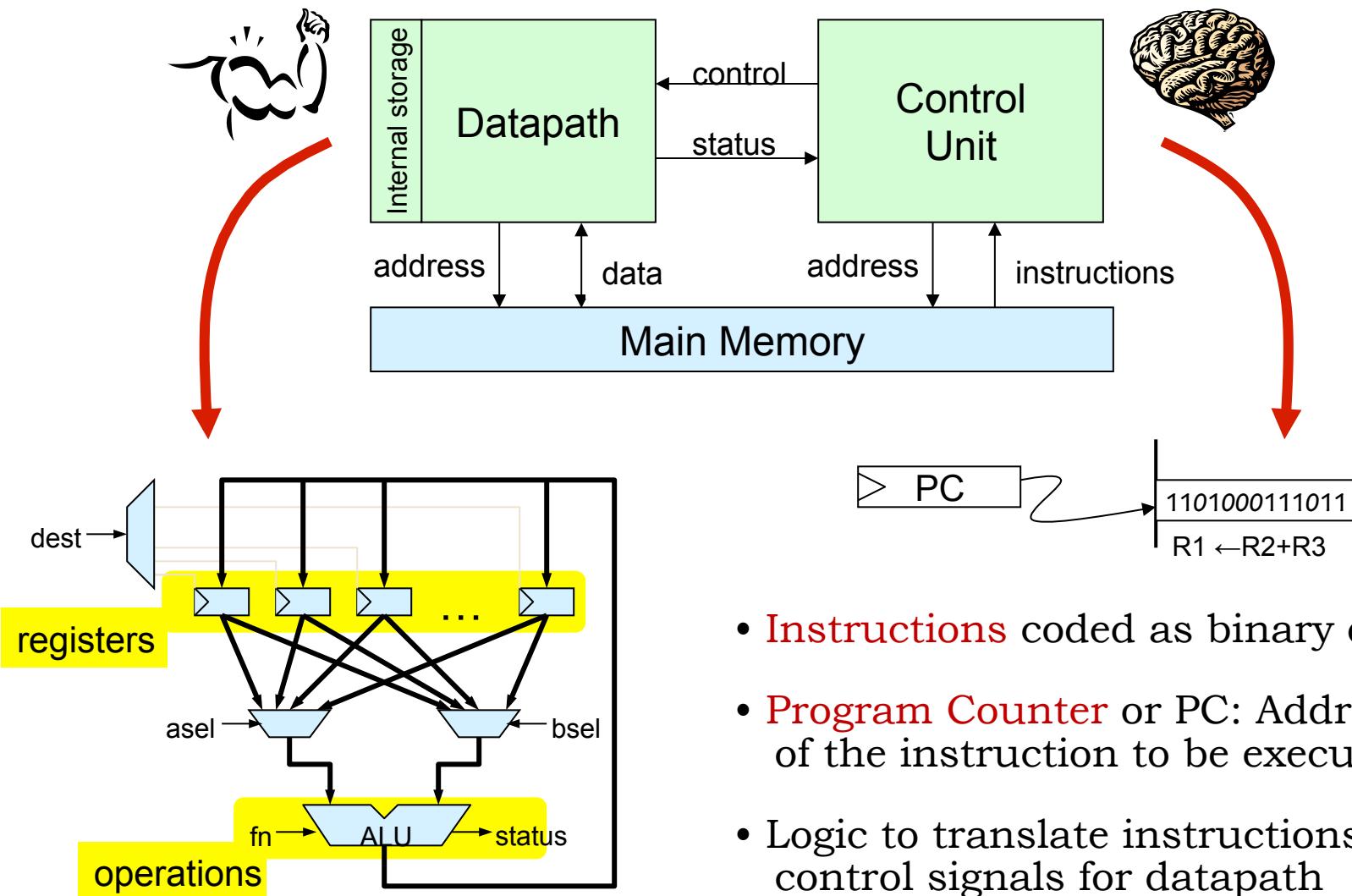
- Central processing unit:
Performs operations on values in registers & memory
- Main memory:
Array of W words of N bits each
- Input/output devices to communicate with the outside world

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



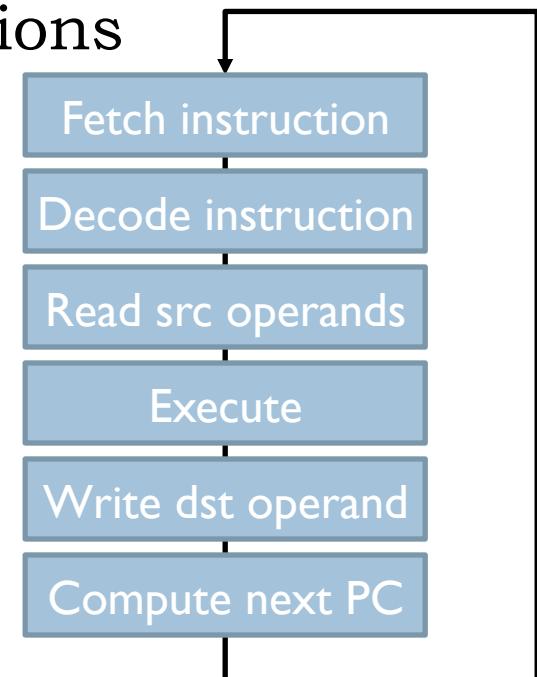
Anatomy of a von Neumann Computer



- **Instructions** coded as binary data
- **Program Counter** or **PC**: Address of the instruction to be executed
- Logic to translate instructions into control signals for 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



Instruction Set Architecture (ISA)

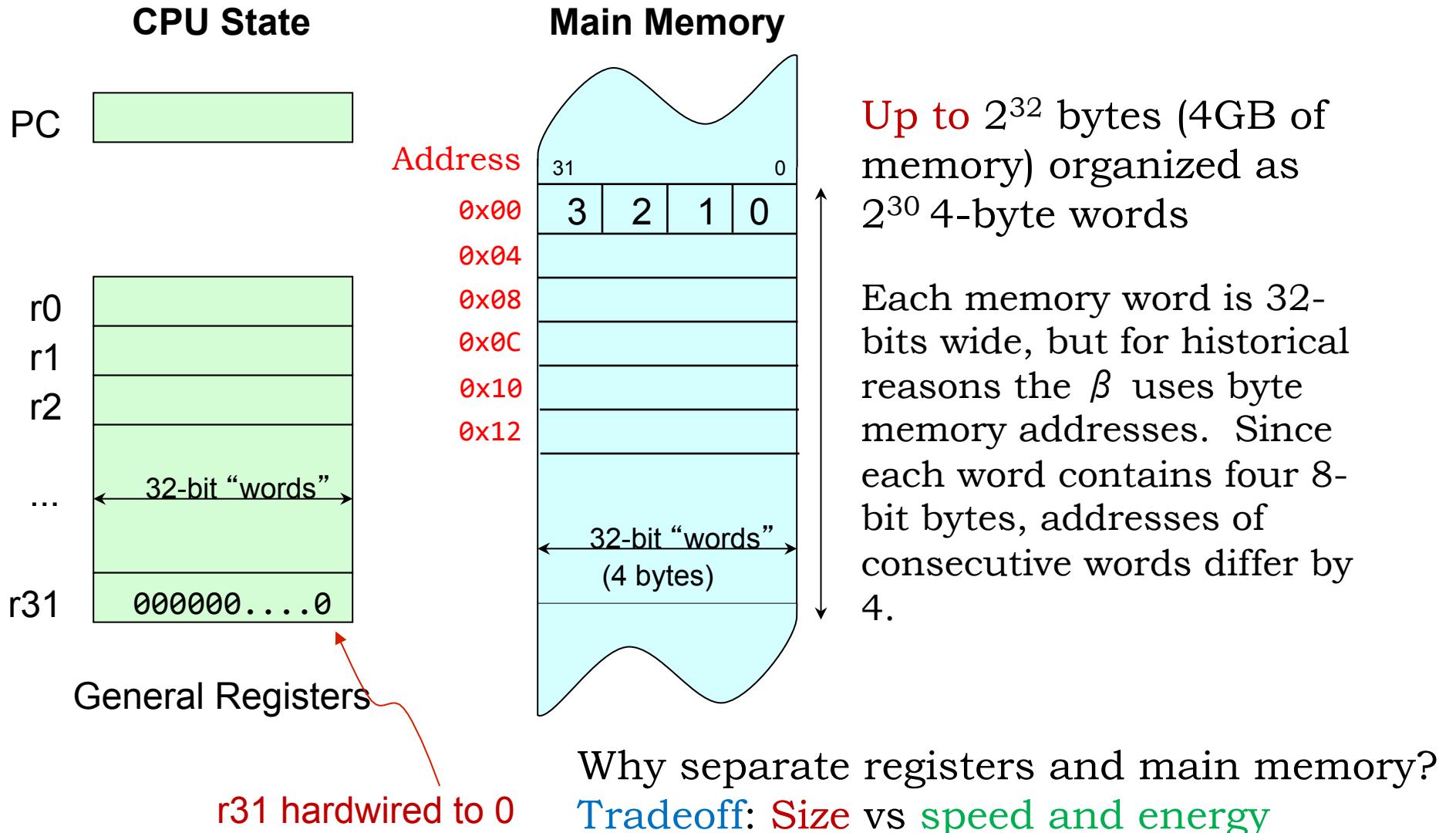
- ISA: The contract between software and hardware
 - Functional definition of **operations** and **storage locations**
 - **Precise description** of how software can invoke and access them
- The ISA is a new layer of abstraction:
 - ISA specifies **what** the hardware provides, **not how** it's implemented
 - 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

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**

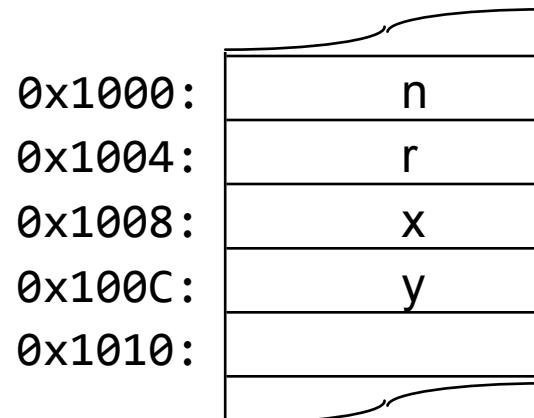
Let's design our own instruction set: the Beta!

Beta ISA: Storage



Storage Conventions

- Variables live in memory
- Registers hold temporary values
- To operate with memory variables
 - Load them
 - Compute on them
 - Store the results



```
int x, y;  
y = x * 37;  
  
R0 ← Mem[0x1008]  
R0 ← R0 * 37  
Mem[0x100C] ← R0
```

Beta ISA: Instructions

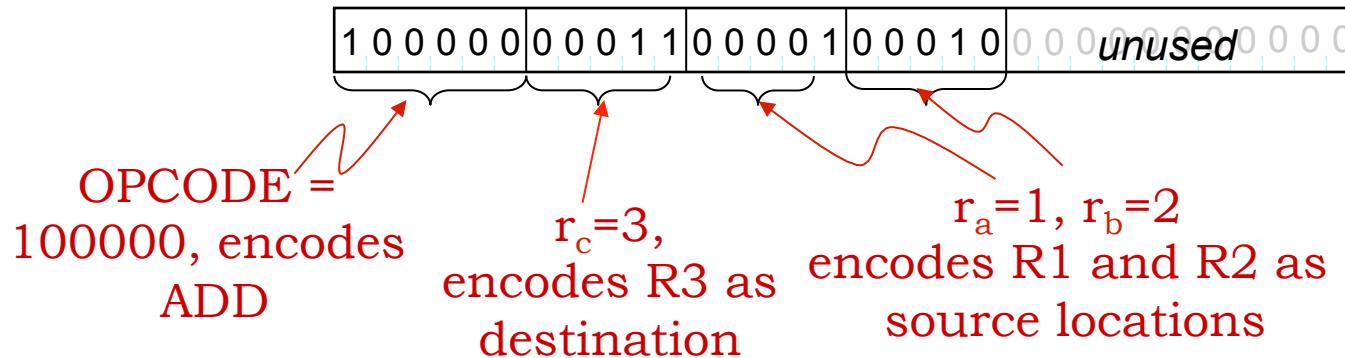
- Three types of instructions:
 - Arithmetic and logical: Perform operations on general registers
 - Loads and stores: Move data between general registers and main memory
 - Branches: Conditionally change the program counter
- All instructions have a fixed length: 32 bits (4 bytes)
 - **Tradeoff** (vs variable-length instructions):
 - Simpler decoding logic, next PC is easy to compute
 - Larger code size

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):

$$\text{Reg}[r_c] \leftarrow \text{Reg}[r_a] + \text{Reg}[r_b]$$

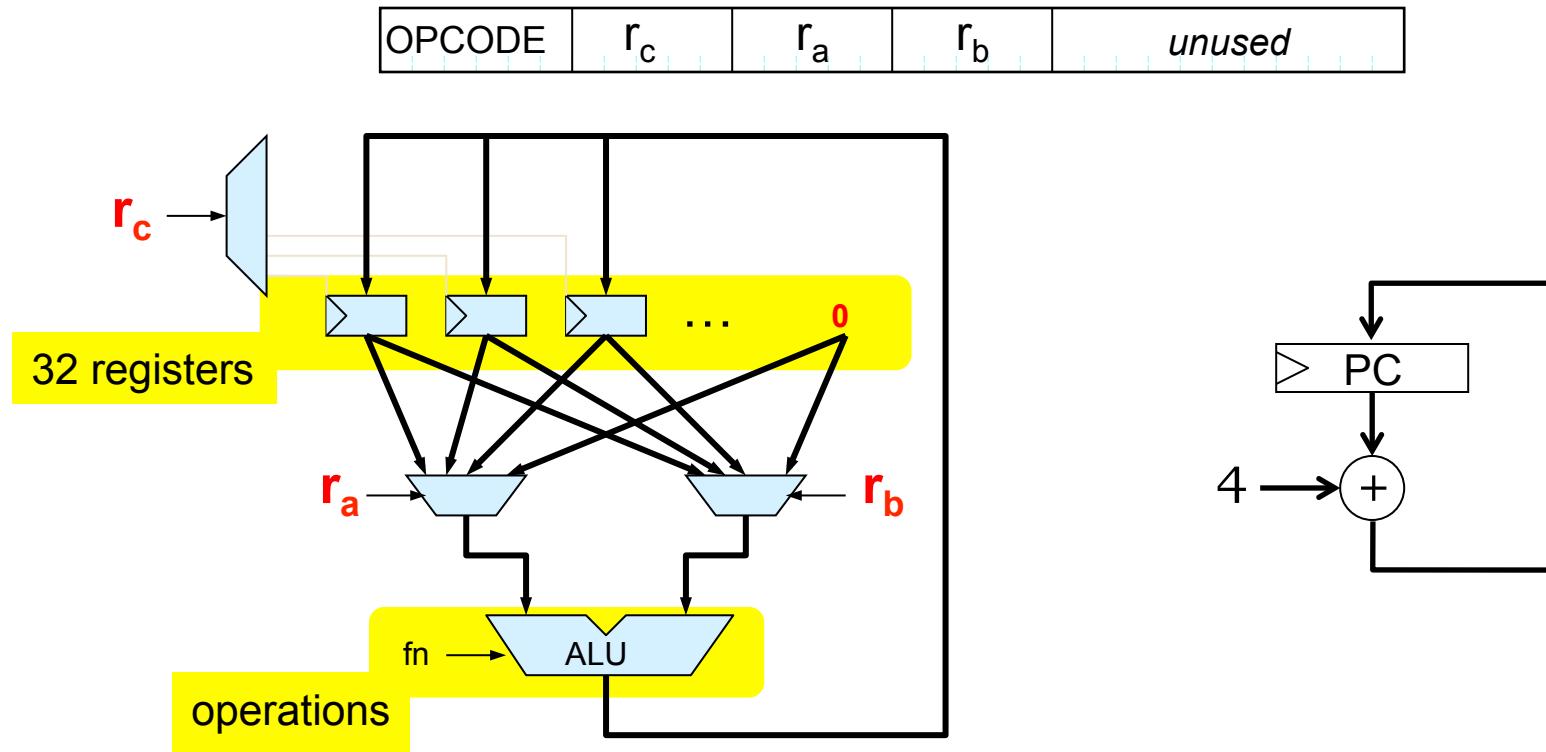
“Add the contents of r_a to the contents of r_b ; store the result in r_c ”

Similar instructions for other ALU operations:

arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR, XNOR
shift: SHL, SHR, SAR

Implementation Sketch #1

Now that we have our first set of instructions, we can create a more concrete implementation sketch:



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

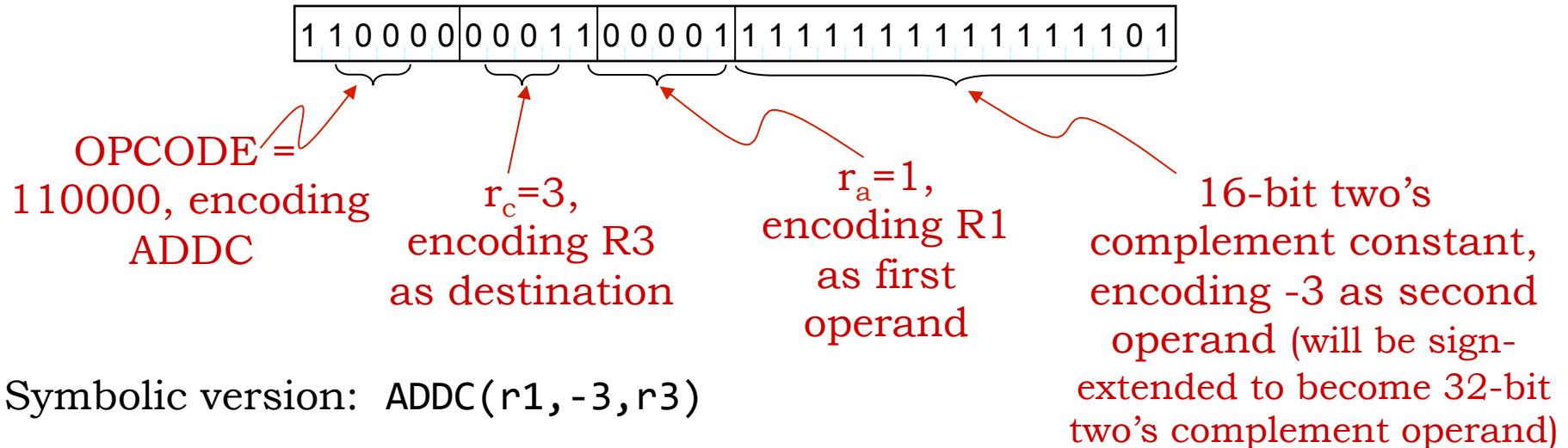
- >50% of executed arithmetic instructions
 - *Loop increments, scaling indices*
- >80% of executed compare instructions
 - *Loop termination condition*
- >25% of executed load instructions
 - *Offsets into data structures*

Beta ALU Instructions with Constant

Format:

OPCODE	r_c	r_a	16-bit signed constant
--------	-------	-------	------------------------

Example instruction: ADDC adds register contents and constant:



Symbolic version: $\text{ADDC}(r1, -3, r3)$

$\text{ADDC}(ra, \text{const}, rc)$:

$\text{Reg}[rc] \leftarrow \text{Reg}[ra] + \text{sext}(\text{const})$

“Add the contents of ra to const ; store the result in rc ”

Similar instructions for other ALU operations:

arithmetic: ADDC, SUBC, MULC, DIVC

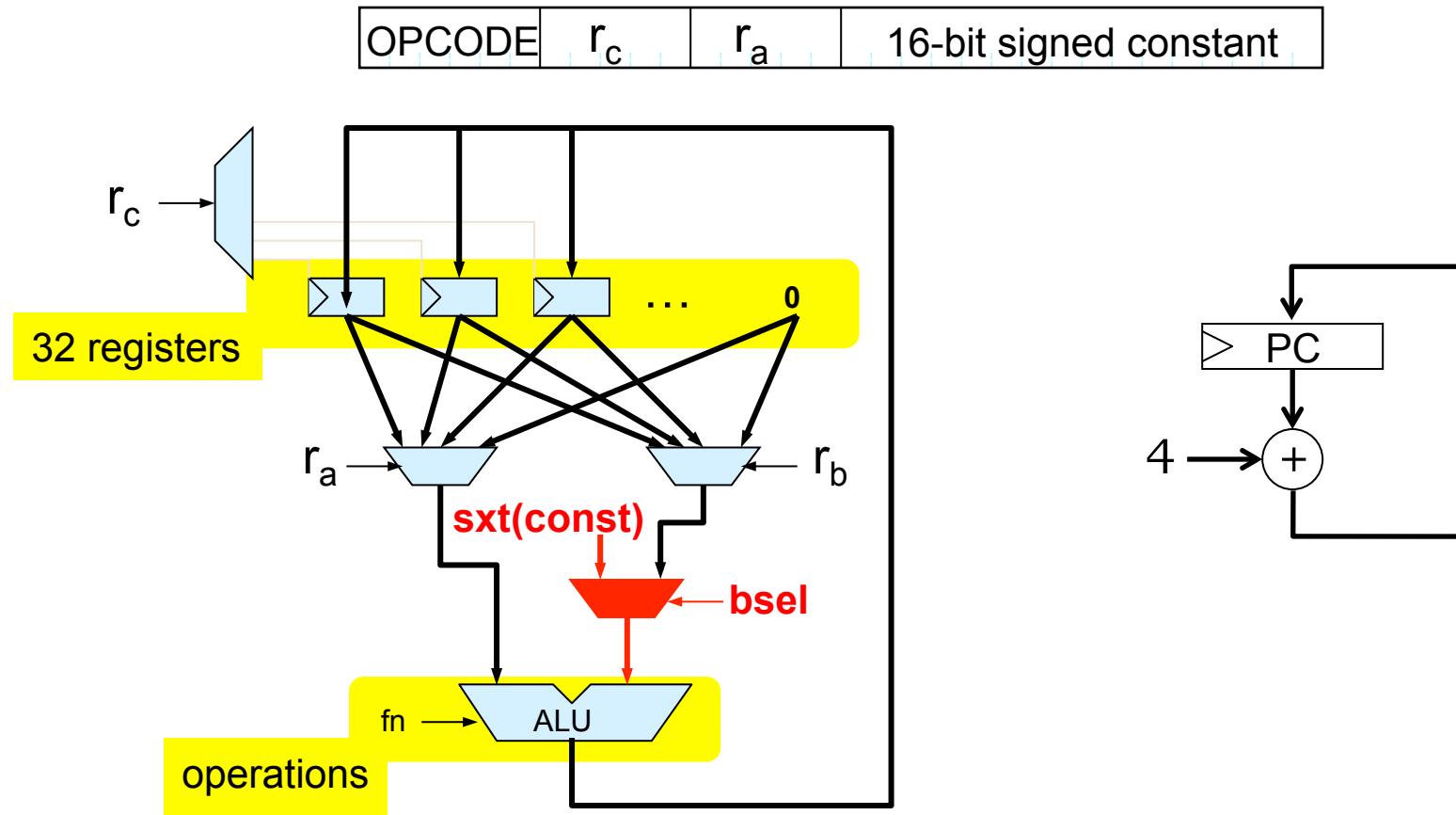
compare: CMPEQC, CMPLTC, CMPLEC

boolean: ANDC, ORC, XORC, XNORC

shift: SHLC, SHRC, SARC

Implementation Sketch #2

Next we add the datapath hardware to support small constants as the second ALU operand:



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!

$LD(ra, const, rc) \quad \text{Reg}[rc] \leftarrow \text{Mem}[\text{Reg}[ra] + \text{sext}(const)]$

Load rc with the contents of the memory location

$ST(rc, const, ra) \quad \text{Mem}[\text{Reg}[ra] + \text{sext}(const)] \leftarrow \text{Reg}[rc]$

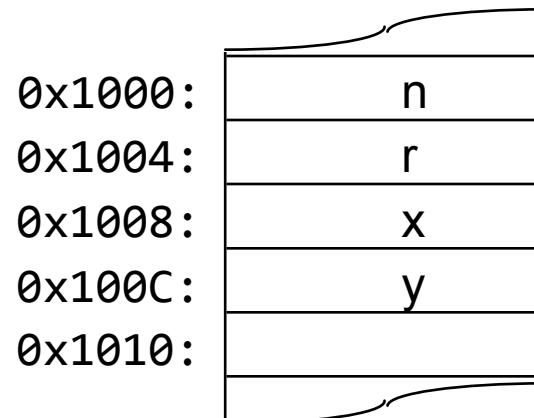
Store the contents of rc into the memory location

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.

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



```
int x, y;  
y = x * 37;
```



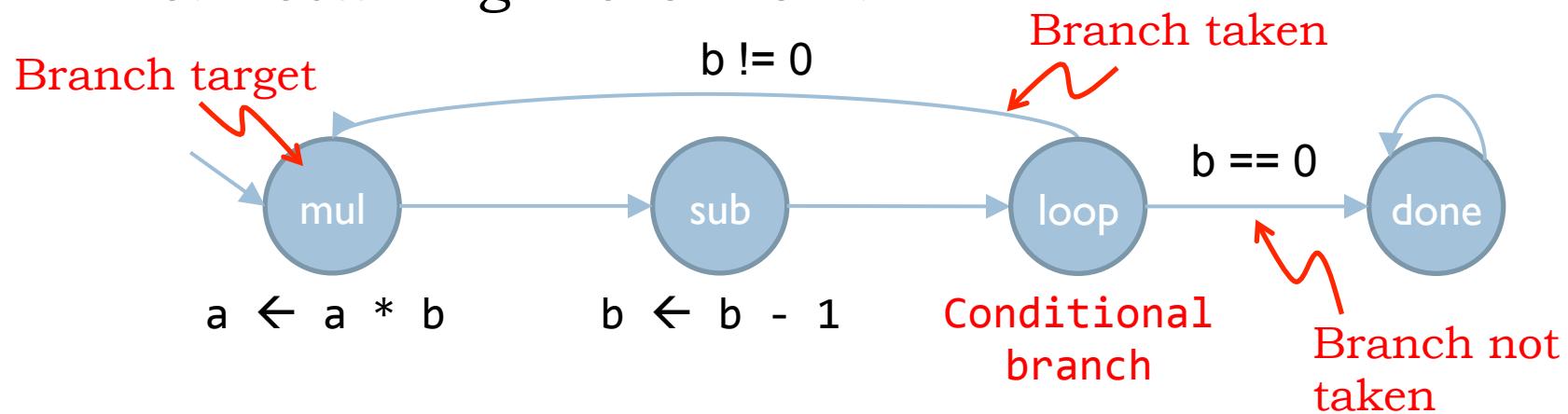
```
R0 ← Mem[0x1008]  
R0 ← R0 * 37  
Mem[0x100C] ← R0
```



```
LD(R31, 0x1008, R0)  
MULC(R0, 37, R0)  
ST(R0, 0x100C, R31)
```

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.

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 ← PC + 4  
Reg[rc] ← NPC  
if (Reg[ra] == 0)  
    PC ← NPC + 4*offset  
else  
    PC ← NPC
```

```
NPC ← PC + 4  
Reg[rc] ← NPC  
if (Reg[ra] != 0)  
    PC ← NPC + 4*offset  
else  
    PC ← NPC
```

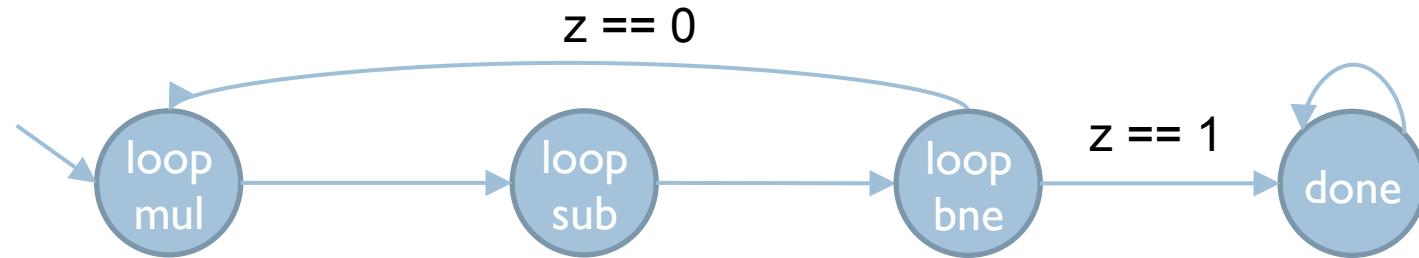
offset = distance in words to branch target, counting from the instruction following the BEQ/BNE. Range: -32768 to +32767.

Can We Solve Factorial Now?

```
int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
} while (b != 0)
```

	<i>// Assume r1 = N</i>
	ADDC(r31, 1, r0) // r0 = 1
	L:MUL(r0, r1, r0) // r0 = r0 * r1
	SUBC(r1, 1, r1) // r1 = r1 - 1
	BNE(r1, L, r31) // if r1 != 0, run MUL next
	// 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

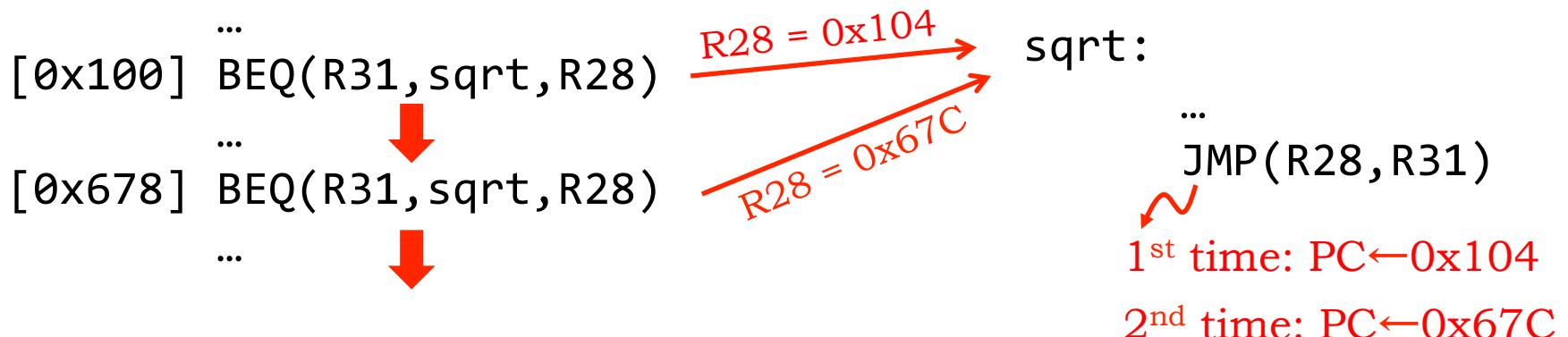
Beta JMP Instruction

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.



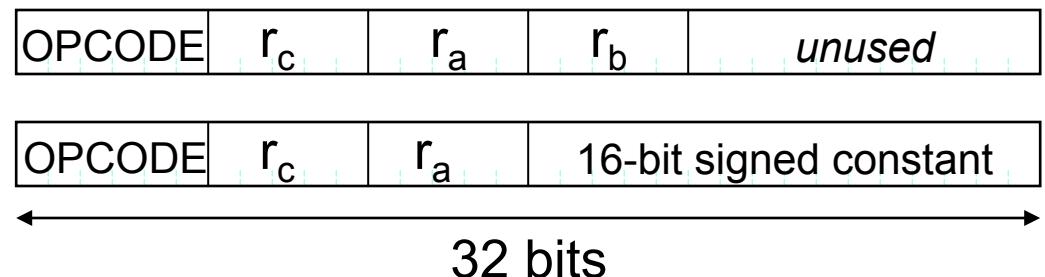
JMP(Ra,Rc): $\text{Reg}[Rc] \leftarrow \text{PC} + 4$
 $\text{PC} \leftarrow \text{Reg}[Ra]$

Useful for procedure call return...



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

1. Basics of Information

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS

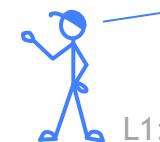
What is “Information”?

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”



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

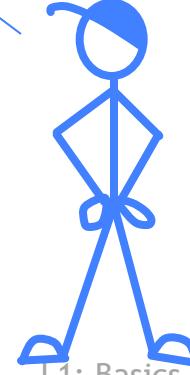
Information received when learning that choice was x_i :

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

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)



Information Conveyed by Data

Even when data doesn't resolve all the uncertainty

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

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

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

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:

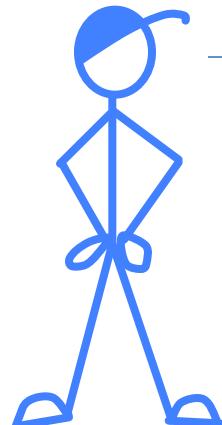
$$N = 36 \quad M = 1 \quad \text{Info content} = \log_2(36/1) = 5.17$$



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.*

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

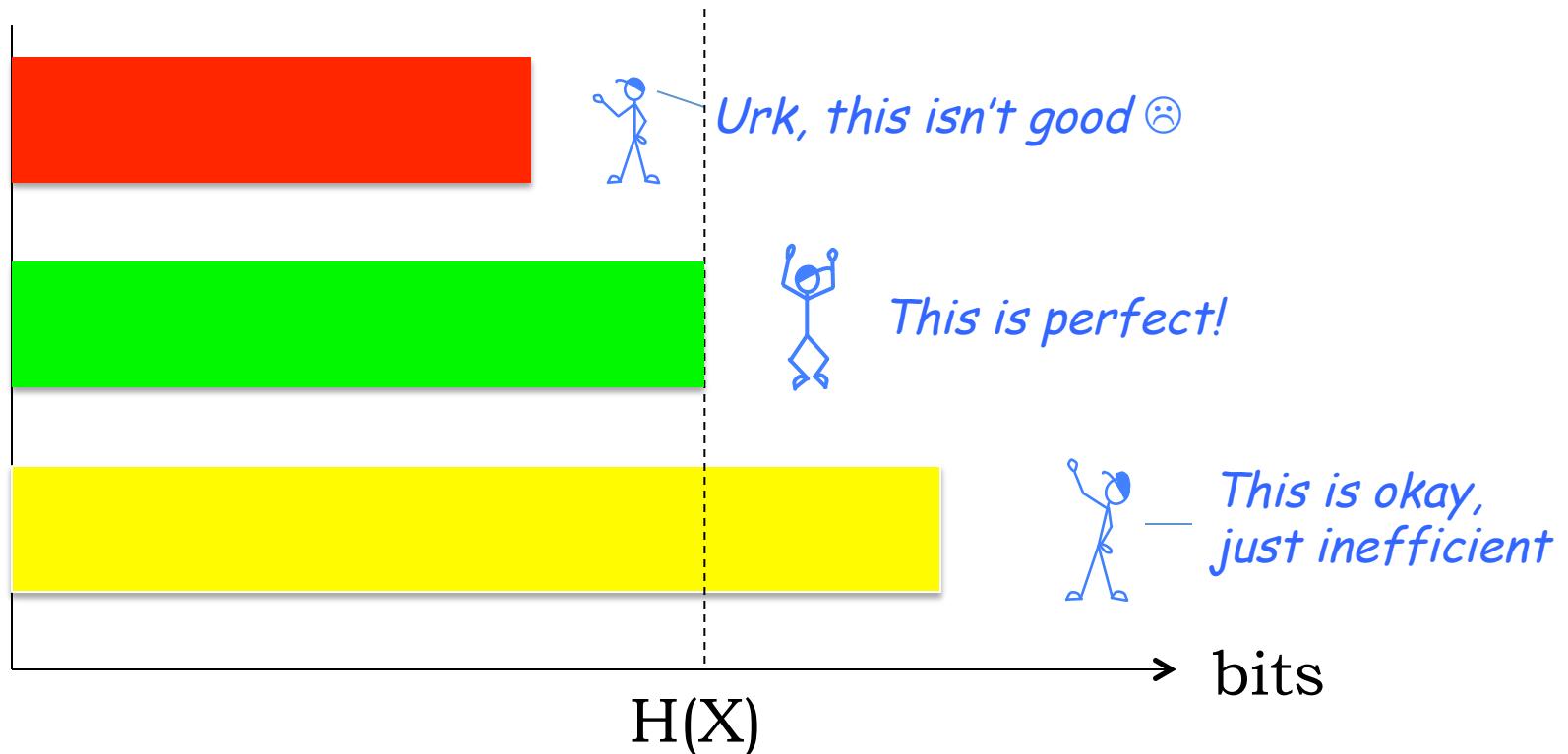
$choice_i$	p_i	$\log_2(1/p_i)$
“A”	1/3	1.58 bits
“B”	1/2	1 bit
“C”	1/12	3.58 bits
“D”	1/12	3.58 bits

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

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



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
0	1	10	11	0 1 1 0  ABBA? ABC? ADA?

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.

Encoding

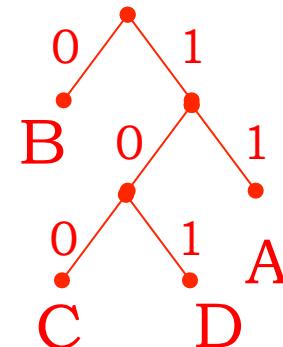
$B \leftrightarrow 0$

$A \leftrightarrow 11$

$C \leftrightarrow 100$

$D \leftrightarrow 101$

Binary tree

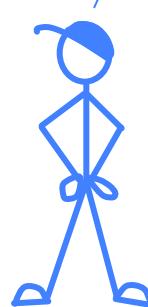
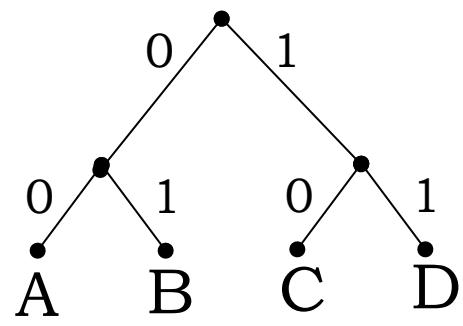


BAA

/ 01111

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

$$\sum_{i=1}^N \left(\frac{1}{N}\right) \log_2 \left(\frac{1}{\frac{1}{N}}\right) = \log_2(N)$$

Examples:

Fixed-length are often a little inefficient...



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

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

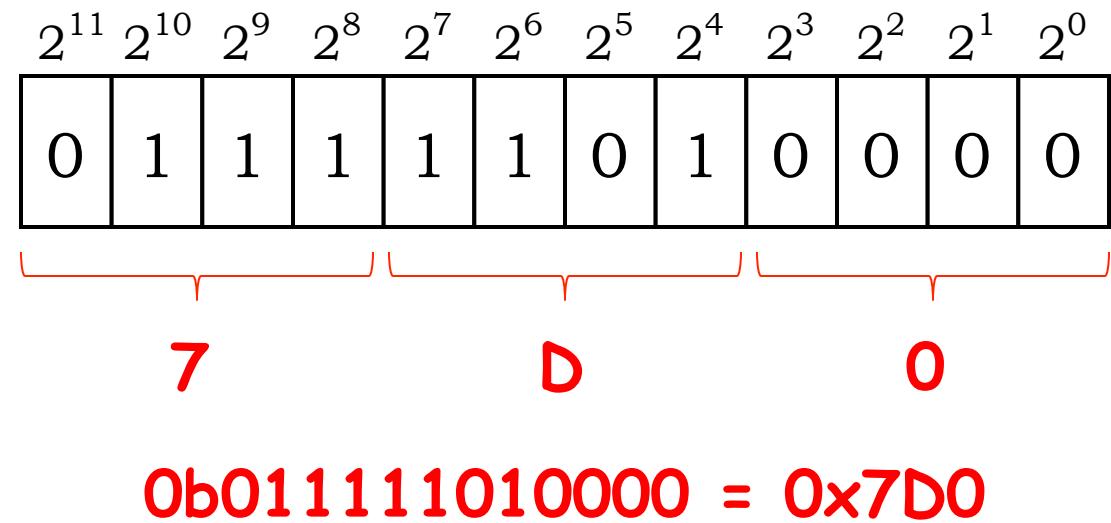
Largest number: $2^N - 1$

Hexadematical 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.

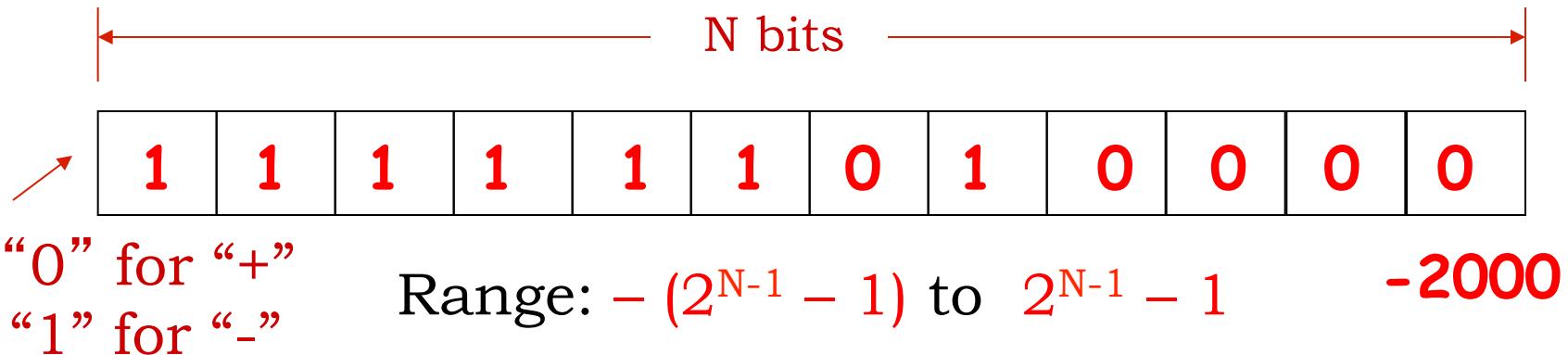
Hexadecimal - base 16	
0000	- 0
0001	- 1
0010	- 2
0011	- 3
0100	- 4
0101	- 5
0110	- 6
0111	- 7
1000	- 8
1001	- 9
1010	- A
1011	- B
1100	- C
1101	- D
1110	- E
1111	- F



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).

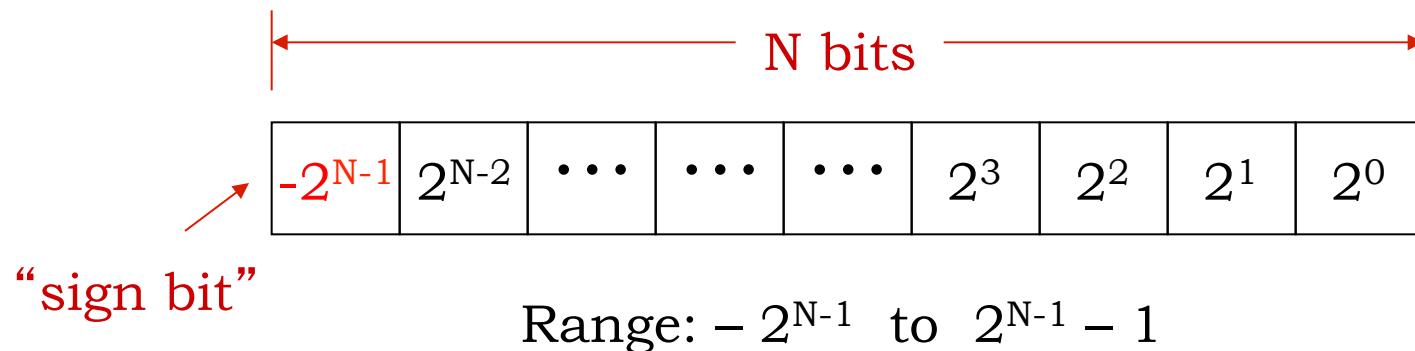
We could adopt that approach for binary representations:



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

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: $00\dots0000$ $\underline{0}$

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!

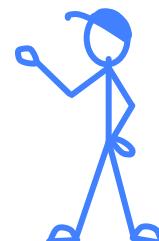
- To compute $B-A$, we'll just use addition and compute $B+(-A)$. But how do we figure out the representation for $-A$?

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

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

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

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



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:

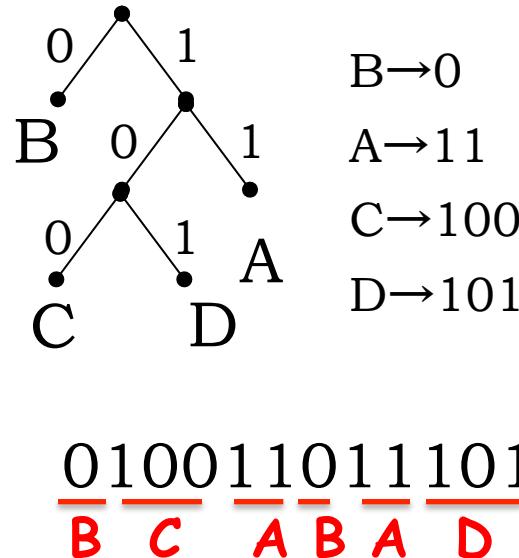
- Higher probability → shorter encodings
- Lower probability → longer encodings

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

Example

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

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



High probability,
Less information

Low probability,
More information

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

Huffman's Algorithm

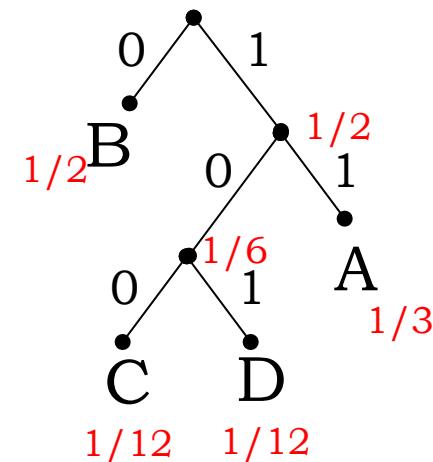
Given a set of symbols and their probabilities, constructs an optimal variable-length encoding.

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



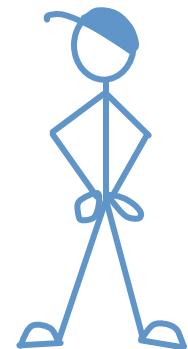
Can We Do Better?

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

Error Detection and Correction

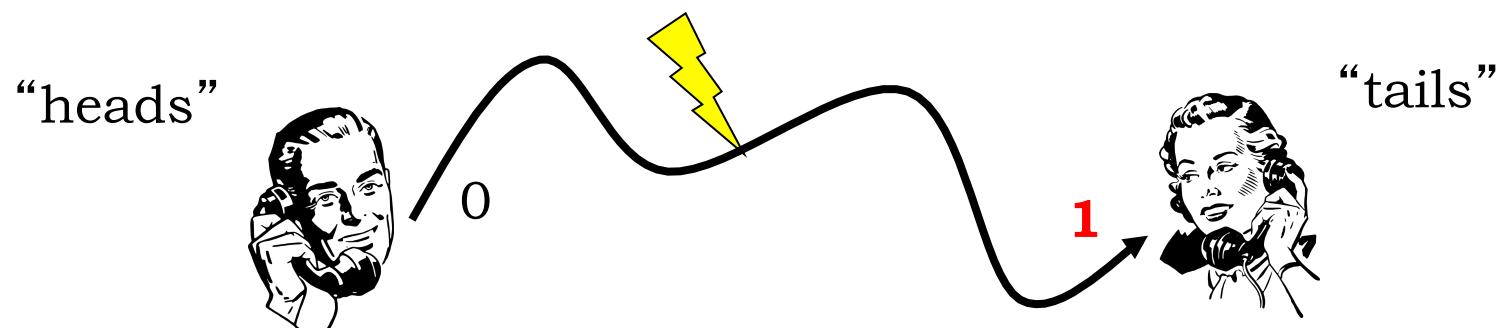
Suppose we wanted to reliably transmit the result of a single coin flip:

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”.



Hamming Distance

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

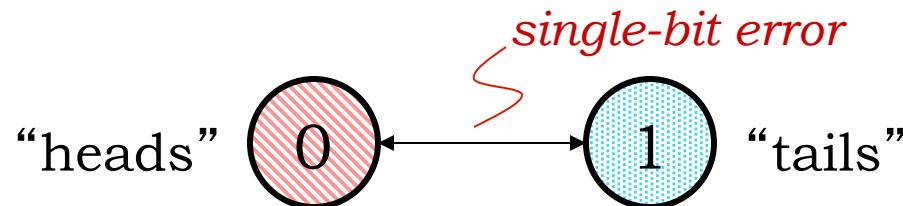
01	1	00	10
↑↑			
01	0	01	10



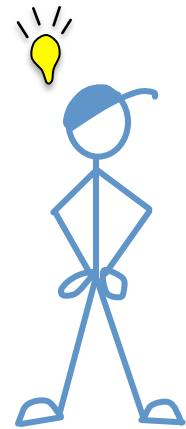
Hamming Distance & Bit Errors

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

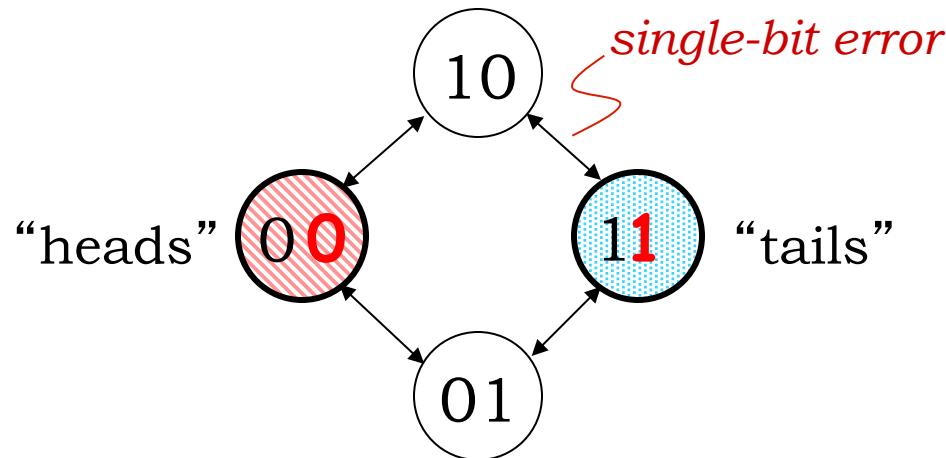
The problem with our simple encoding is that the two valid code words (“0” and “1”) also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word...



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 $\min \text{HD}(\text{code words} + \text{parity}) = 2$.

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

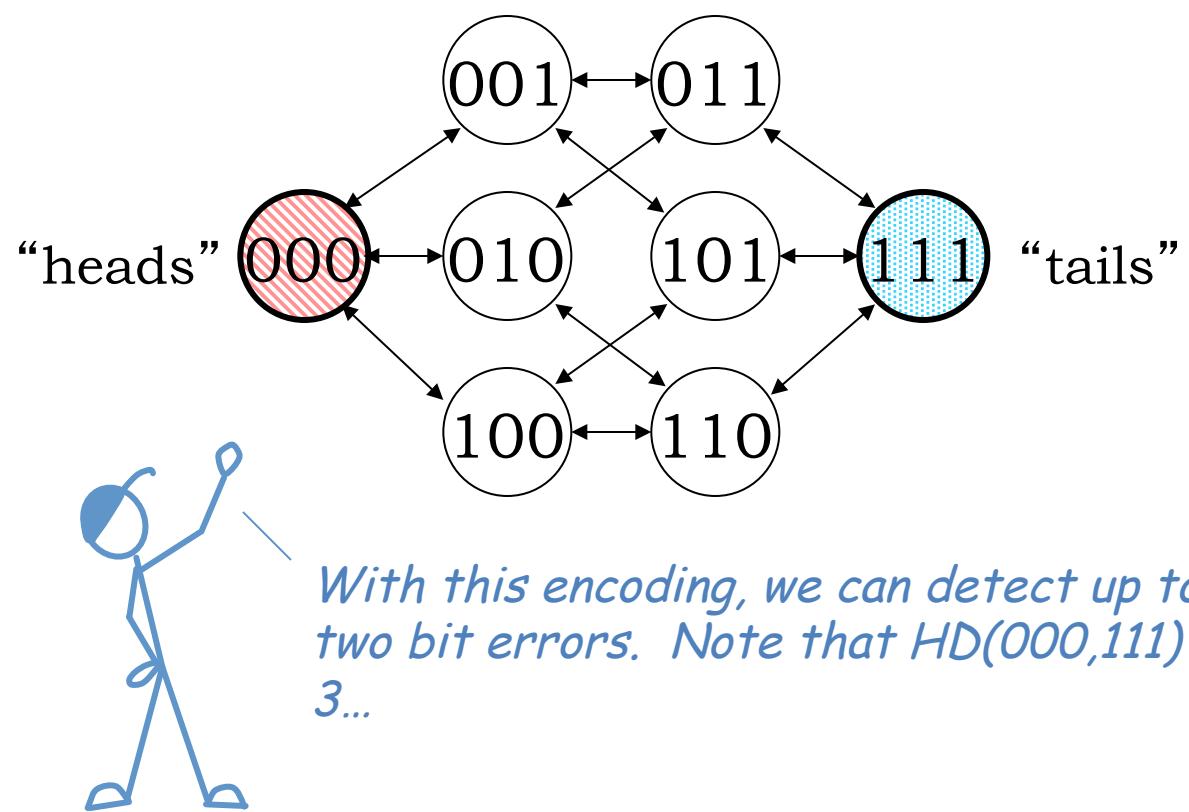
0 1 1 0 0 **0** 0 1 0 0 1 1 → single-bit error (detected)

0 1 1 0 0 **0** **1** 1 0 0 1 1 → 2-bit error (not detected)

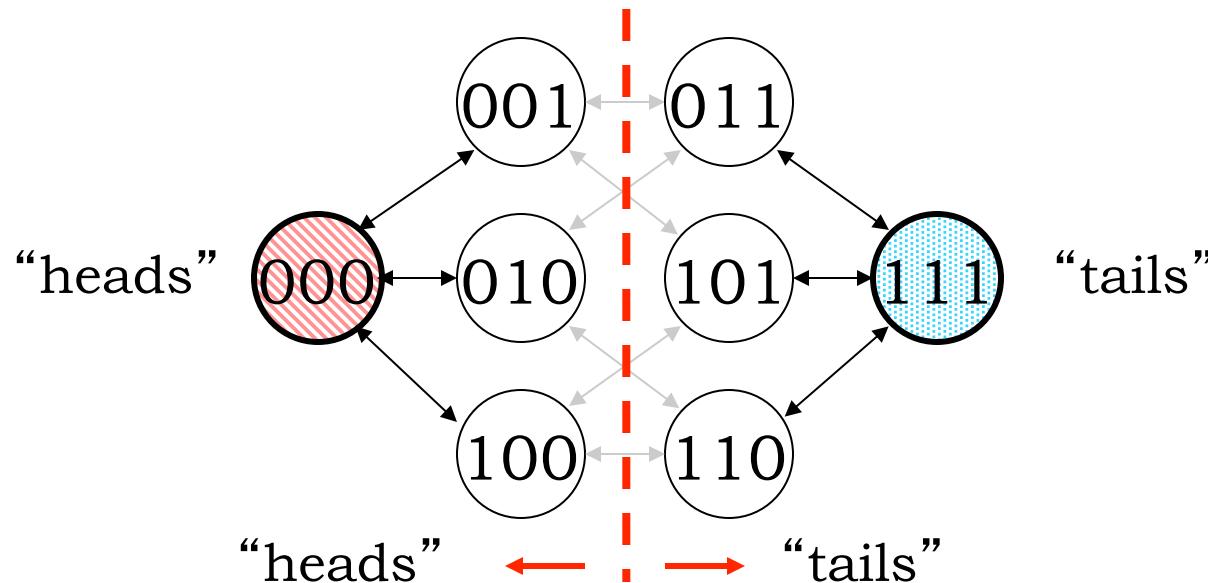
- One can “count” by summing the bits in the word modulo 2 (which is equivalent to XOR’ing the bits together).

Detecting Multi-bit Errors

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



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.

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

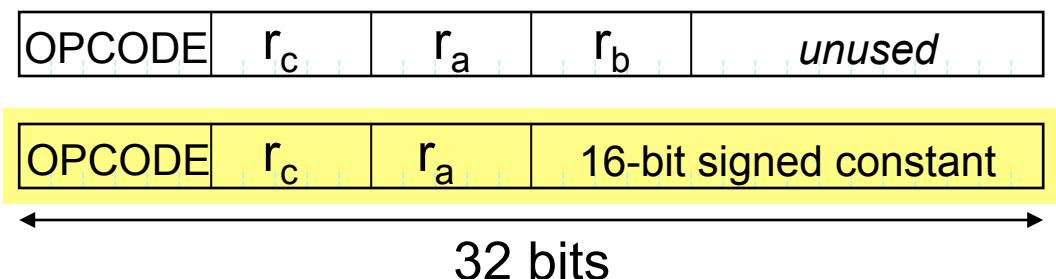
10. Assembly Language, Models of Computation

6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2015 MIT EECS

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

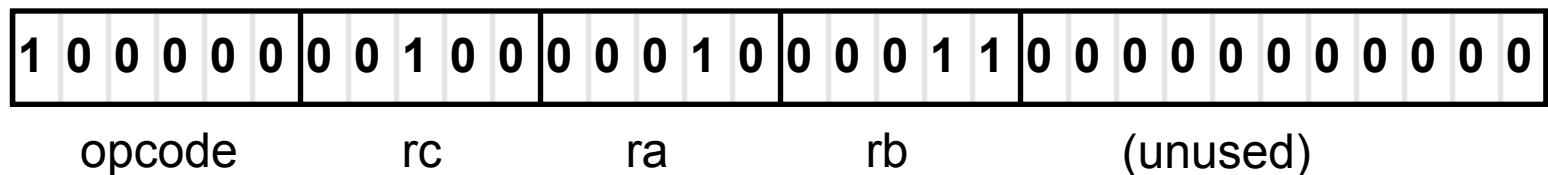


- Instruction formats:

- Instruction classes:
 - ALU: Two input registers, or register and constant
 - Loads and stores: access memory
 - Branches, Jumps: change program counter

Programming Languages

32-bit (4-byte) ADD instruction:



Means, to the BETA, $\text{Reg}[4] \leftarrow \text{Reg}[2] + \text{Reg}[3]$

We'd rather write in *assembly language*:

ADD(R2, R3, R4)

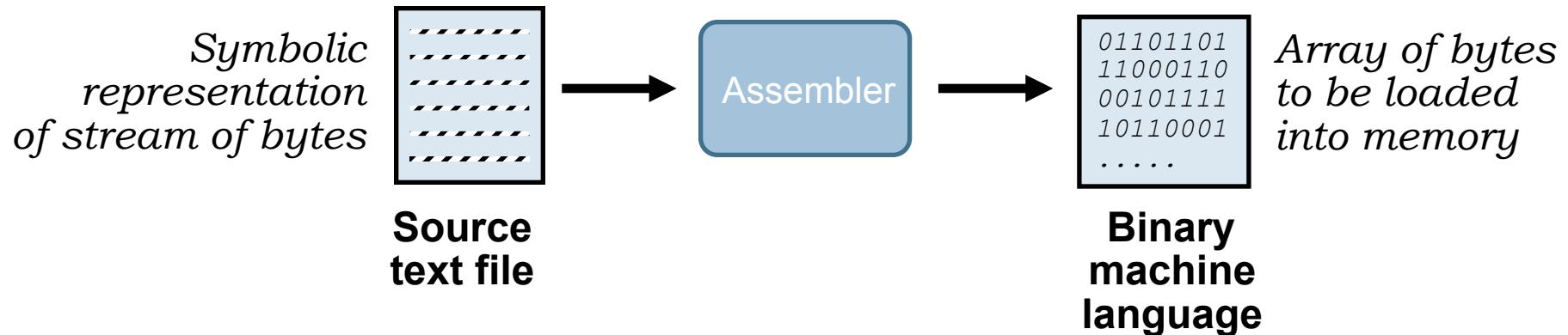
Today

or better yet a *high-level language*:

a = b + c;

Coming up

Assembly Language



- Abstracts bit-level representation of instructions and addresses
- We'll learn UASM ("microassembler"), built into BSim
- Main elements:
 - Values
 - Symbols
 - Labels (symbols for addresses)
 - Macros

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

How Does It Get Assembled?

Text input

N = 12

ADDC(r31, N, r1)

ADDC(r31, 1, r0)

loop: MUL(r0, r1, r0)

SUBC(r1, 1, r1)

BNE(r1, loop, r31)

- Load predefined symbols into a symbol table
- Read input line by line
 - Add symbols to symbol table as they are defined
 - Expand macros, translating symbols to values first

Binary output

→ 110000 00001 11111 00000000 00001100 [0x00]

110000 00000 11111 00000000 00000001 [0x04]

100010 00000 00000 00001 0000000000 [0x08]

...

Symbol table

Symbol	Value
r0	0
r1	1
...	
r31	31
N	12
loop	8

Registers are Predefined Symbols

- $r0 = 0, \dots, r31 = 31$

- Treated like
normal symbols:

`ADDC(r31, N, r1)`



Substitute symbols with their values

`ADDC(31, 12, 1)`



Expand macro

`110000 00001 11111 00000000 00001100`

- No “type checking” if you use the wrong opcode...

`ADDC(r31, r12, r1)`



`ADDC(31, 12, 1)`

$\text{Reg}[1] \leftarrow \text{Reg}[31] + 12$

`ADD(r31, N, r1)`



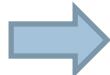
`ADD(31, 12, 1)`

$\text{Reg}[1] \leftarrow \text{Reg}[31] + \text{Reg}[12]$

Labels and Offsets

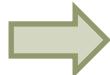
Input file

```
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]  
011101 11111 00001 11111111 11111101 [0x10]
```



$$\begin{aligned}\text{offset} &= (\text{label} - \text{addr of BNE/BEQ})/4 - 1 \\ &= (8 - 16)/4 - 1 = -3\end{aligned}$$

- **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

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

```
⇒ 37 37+1 37+2 37+3 ⇒ 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:

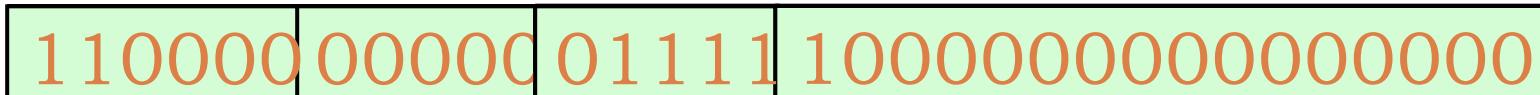
Mem: 0x100 0x101 0x102 0x103
 0xef 0xbe 0xad 0xde



Boy, that's hard to read.
Maybe, those big-endian
types do have a point.

Assembly of Instructions

-32768 = 10000000000000000000000000000000



```
// 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 betaopc(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) betaopc(OP,RA,((LABEL- (.+4))>>2),RC)
```

“.align 4” ensures instructions will begin on word boundary (i.e., address = 0 mod 4)

For example:

.macro ADDC(RA,C,RC) betaopc(0x30,RA,C,RC)

ADDC(R15, -32768, R0) --> betaopc(0x30,15,-32768,0)

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

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...

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)
```

Factorial with Pseudoinstructions

Before

```
N = 12  
ADDC(r31, N, r1)  
ADDC(r31, 1, r0)  
loop: MUL(r0, r1, r0)  
SUBC(r1, 1, r1)  
BNE(r1, loop, r31)
```

After

```
N = 12  
CMOVE(N, r1)  
CMOVE(1, r0)  
loop: MUL(r0, r1, r0)  
SUBC(r1, 1, r1)  
BNE(r1, loop)
```

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)

Symbol table

Symbol	Value
...	
N	0
factN	4

LD(r31, N, r1)



LD(31, 0, 1)

Reg[1] ← Mem[Reg[31] + 0]

← Mem[0]

← 12

UASM Expressions and Layout

- Values can be written as expressions
 - Assembler evaluates expressions, they are *not* translated to instructions to compute the value!

```
A = 7 + 3 * 0x0cc41
B = A - 3
```
- 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 = .               // Symbol “k” has value 0x104
LONG(0x00dec0de)
. = .+16            // Skip 16 bytes
LONG(0xc0ffeeee)
```

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

Universality?

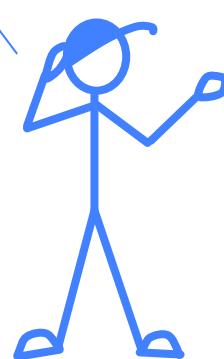
- Recall: We say a set of Boolean gates is universal if we can implement any Boolean function using only gates from that set.
- What problems can we solve with a von Neumann computer? (e.g., the Beta)
 - Everything that FSMs can solve?
 - Every problem?
 - Does it depend on the ISA?
- Needed: a mathematical model of computation
 - Prove what can be computed, what can't

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...

*We've got FSMs...
what else do we need?*

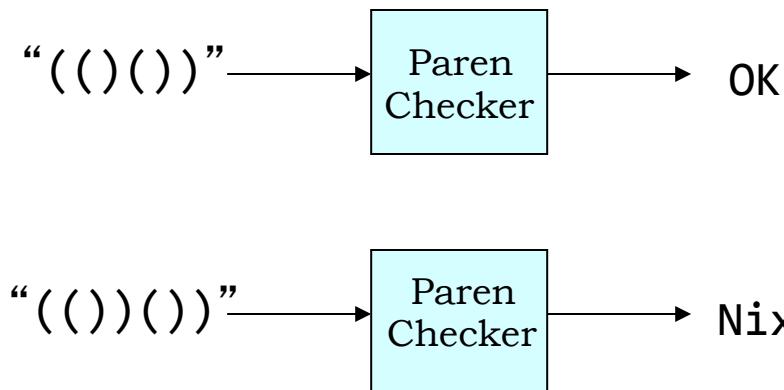


- switches
- gates
- combinational logic
- memories
- FSMs

Are FSMs the ultimate digital computing 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:

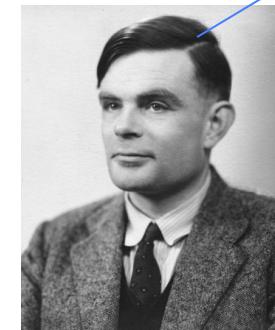
Given any string of coded left & right parens, outputs 1 if it is balanced, else 0.

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

Turing Machines

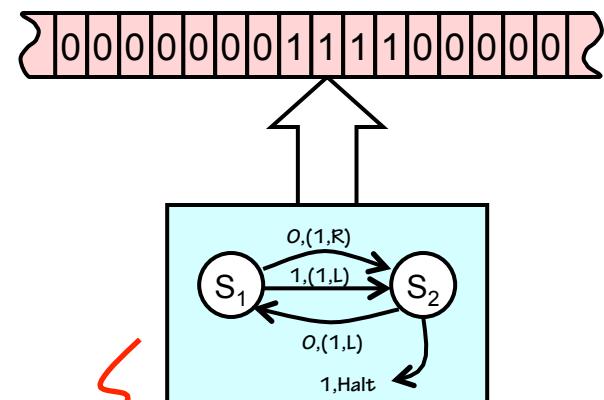
Alan Turing was one of a group of researchers studying alternative models of computation.

He proposed a conceptual model consisting of an FSM combined with an infinite digital tape that could be read and written at each step.

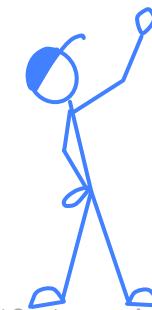
- encode input as symbols on tape
- FSM reads tape/writes symbols/ changes state until it halts
- Answer encoded on tape

Turing's model (like others of the time) solves the "FINITE" problem of FSMs.

Bounded tape configuration can be expressed as a (large!) integer



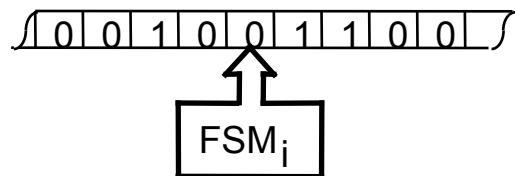
FSMs can be enumerated and given a (very large) integer index.



We can talk about TM 347 running on input 51, producing an answer of 42.
TMs as integer functions:
 $y = TM_I[x]$

Other Models of Computation...

Turing Machines [Turing]



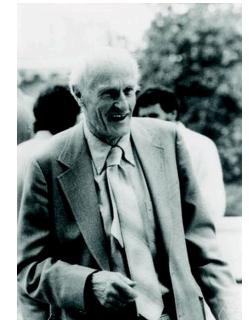
Alan Turing

Recursive Functions [Kleene]

$$F(0, x) \equiv x$$

$$F(1+y, x) \equiv 1+F(x, y)$$

```
(define (fact n)
  (... (fact (- n 1)))
```



Stephen Kleene

Lambda calculus [Church, Curry, Rosser...]



Alonzo Church

$$\lambda x. \lambda y. xxy$$

```
(lambda (x) (lambda (y) (x (x y))))
```

Production Systems [Post, Markov]



$$\alpha \rightarrow \beta$$

```
IF pulse=0 THEN  
  patient=dead
```

Emile Post

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.

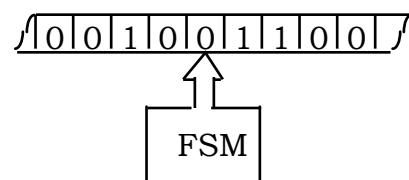
$$f(x) \text{ computable} \Leftrightarrow \text{for some } k, \text{ all } x \\ f(x) = T_k[x]$$



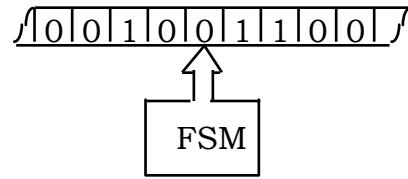
meanwhile...

Turing machines Galore!

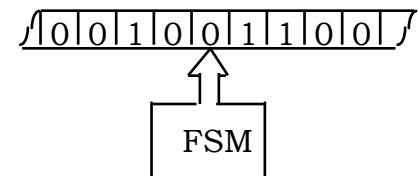
“special-purpose”
Turing Machines...



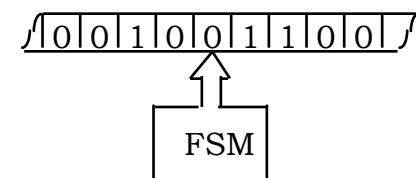
Multiplication



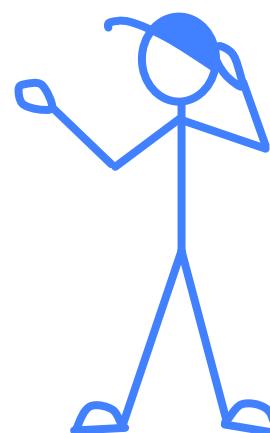
Sorting



Factorization



Primality Test



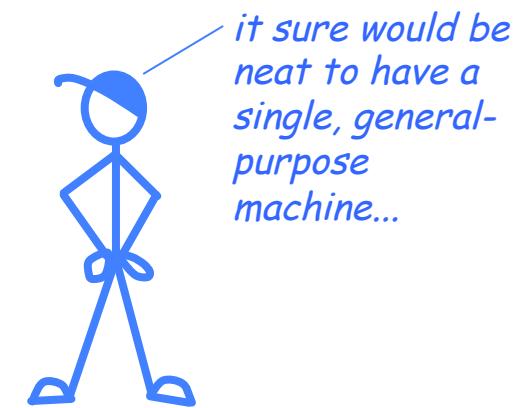
*Is there an alternative to
infinitely many ad-hoc Turing
Machines?*

The Universal Function

Here's an interesting function to explore: the Universal function, U , defined by

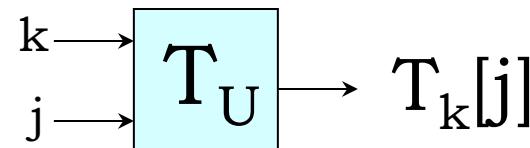
$$U(k, j) = T_k[j]$$

Could this be computable???



*it sure would be
neat to have a
single, general-
purpose
machine...*

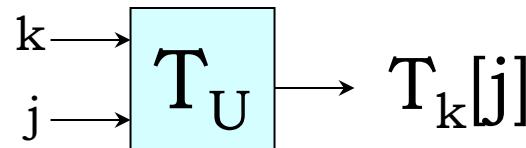
SURPRISE! U is computable by a Turing Machine:



In fact, there are infinitely many such machines. Each 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.

Turing Universality

The *Universal Turing Machine* is the paradigm for modern general-purpose computers!

Basic threshold test: Is your computer *Turing Universal*?

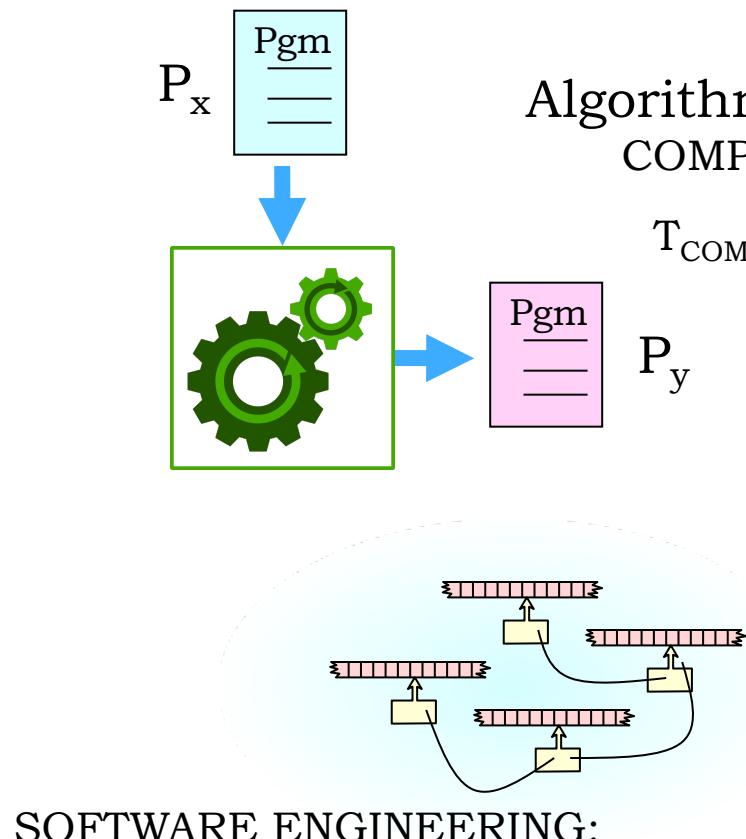
- If so, it can emulate every other Turing machine!
- Thus, your computer can compute any computable function

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.

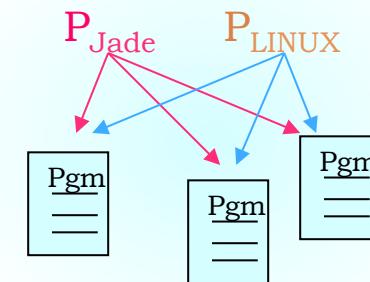
Coded Algorithms: Key to CS

data vs hardware



Algorithms as data: enables
COMPILERS: analyze, optimize, transform behavior

$$T_{\text{COMPILER-X-to-Y}}[P_x] = P_y \text{, such that } T_y[P_x, z] = T_y[P_y, z]$$



LANGUAGE DESIGN: Separate
specification from implementation

- C, Java, JSIM, Linux, ... all run on X86, Sun, ARM, JVM, CLR, ...
- Parallel development paths:
 - Language/Software design
 - Interpreter/Hardware design

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

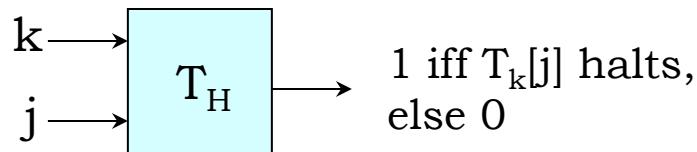
The most famous uncomputable function is the so-called Halting function, $f_H(k, j)$, defined by:

$$\begin{aligned} f_H(k, j) &= 1 \text{ if } T_k[j] \text{ halts;} \\ &\quad 0 \text{ otherwise.} \end{aligned}$$

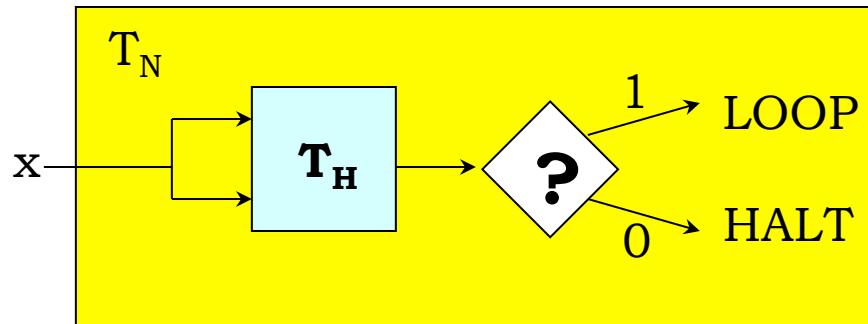
$f_H(k, j)$ determines whether the k^{th} TM halts when given a tape containing j .

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:



$T_N[x]$: LOOPS if $T_x[x]$ halts;
HALTS if $T_x[x]$ loops

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!

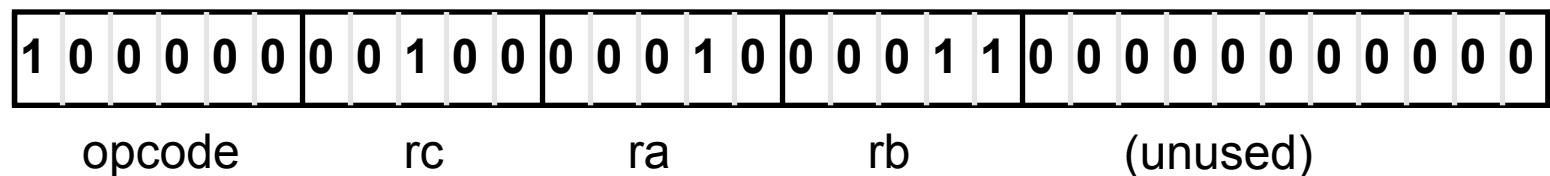
11. Compilers

6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2015 MIT EECS

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)*

High-Level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm:

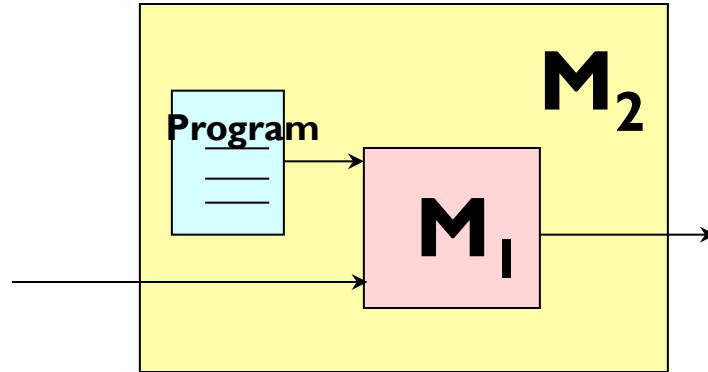
```
/* Compute greatest common divisor
 * using Euclid's method
 */
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```

- 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

Interpretation

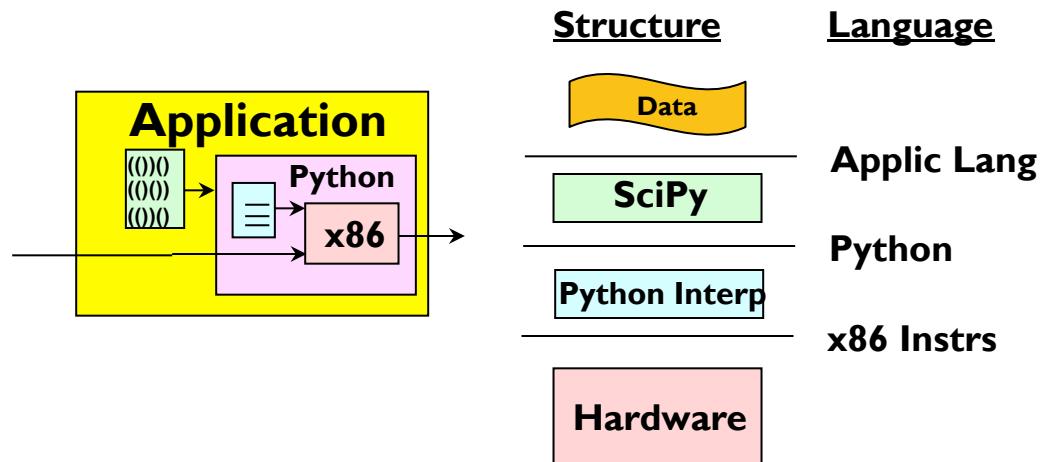
Model of Interpretation:



- Start with some hard-to-program machine, say M_1
- Write a single program for M_1 that mimics the behavior of some easier machine, M_2
- Result: a “virtual” M_2

Layers of interpretation:

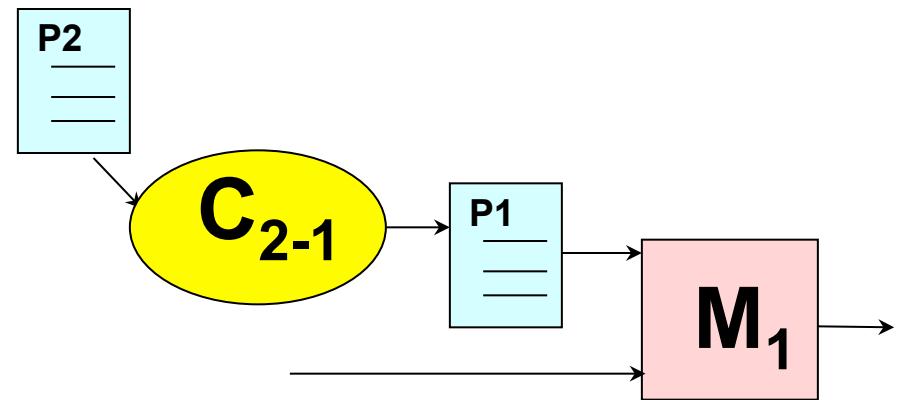
- Often we use several layers of interpretation to achieve desired behavior, e.g.:
- x86 CPU, running
 - Python, running
 - SciPy application, performing
 - Numerical analysis



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!

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?

Compilers

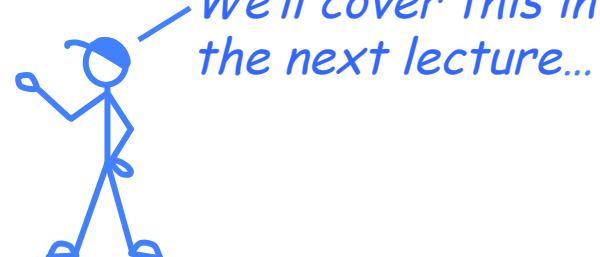
- Bare minimum for a functional compiler:



- 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

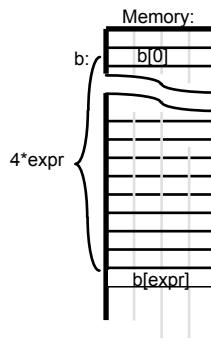
A Simple Compilation Strategy

- 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, ...)`



`compile_expr(expr) ⇒ 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)⇒Rx`
 - `ST(Rx,a)`
- Variables: $b[expr] \Rightarrow Rx$
 - `compile_expr(expr)⇒Rx`
 - `MULC(Rx,bsize,Rx)`
 - `LD(Rx,b,Rx)`
 - ...
 - // reserve array space
 - b: $\dots = \dots + bsize * bLen$
- Operations:
 $expr_1 + expr_2 \Rightarrow Rx$
 - `compile_expr(expr1)⇒Rx`
 - `compile_expr(expr2)⇒Ry`
 - `ADD(Rx,Ry,Rx)`



Compiling Expressions

C code:

```
int x, y;  
y = (x-3)*(y+123456)
```

Beta assembly code:

```
x: LONG(0)  
y: LONG(0)  
c1: LONG(123456)  
...  
LD(x, r1)  
CMOVE(3, r2)  
SUB(r1, r2, r1)
```

```
LD(y, r2)  
LD(c1, r3)  
ADD(r2, r3, r2)  
MUL(r2, r1, r1)  
ST(r1, y)
```

```
compile_expr(y = (x-3)*(y+123456))  
compile_expr((x-3)*(y+123456))  
compile_expr(x-3)  
compile_expr(x)  
LD(x, r1)  
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)
```

compile_statement

- Unconditional: *expr*;

Beta assembly:

`compile_expr(expr)`

- Compound: { *statement*₁; *statement*₂; ... }

Beta assembly:

`compile_statement(statement1)`

`compile_statement(statement2)`

...

compile_statement: Conditional

C code:

```
if (expr)
    statement;
```

Beta assembly:

$\text{compile_expr(expr)} \Rightarrow Rx$

$BF(rx, \text{Lendif})$

$\text{compile_statement(statement)}$

$Lendif:$

C code:

```
if (expr)
    statement1;
else
    statement2;
```

Beta assembly:

$\text{compile_expr(expr)} \Rightarrow Rx$

$BF(rx, \text{Lelse})$

$\text{compile_statement(statement}_1\text{)}$

$BR(Lendif)$

$Lelse:$

$\text{compile_statement(statement}_2\text{)}$

$Lendif:$

compile_statement: Iteration

C code:

```
while (expr)  
    statement;
```

Beta assembly:

```
Lwhile:  
    compile_expr(expr)⇒Rx  
    BF(rx, Lendwhile)  
    compile_statement(statement)  
    BR(Lwhile)  
Lendwhile:
```

Better Beta assembly:

```
BR(Ltest)  
Lwhile:  
    compile_statement(statement)  
Ltest:  
    compile_expr(expr)⇒Rx  
    BT(rx, Lwhile)
```

C code:

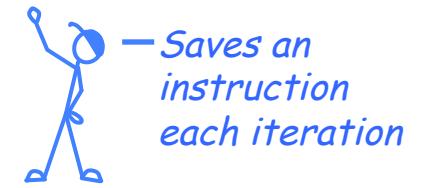
```
for (init; test; increment)  
    statement;
```

Example:

```
for (i=0; i < 10; i = i + 1)  
    sum = sum + b[i];
```

is equivalent to:

```
init;  
while (test) {  
    statement;  
    increment;  
}
```



Putting It All Together: Factorial

```
int n = 20;      { n: LONG(20)
int r = 0;       { r: LONG(0)
start:
r = 1;          {
while (n > 0) { loop:
r = r*n;        {
n = n-1;        {
}                  {
test:
done:           }

LD(r, r3)
LD(n,r1)
MUL(r1, r3, r3)
ST(r3, r)
LD(n,r1)
SUBC(r1, 1, r1)
ST(r1, n)

LD(n, r1)
CMPLT(r31, r1, r2)
BT(r2, loop)
```

Easy translation

Slow code
(10 instructions
in the loop)

Optimization: keep values in regs

```
int n = 20,      n: LONG(20)
int r;          r: LONG(0)
```

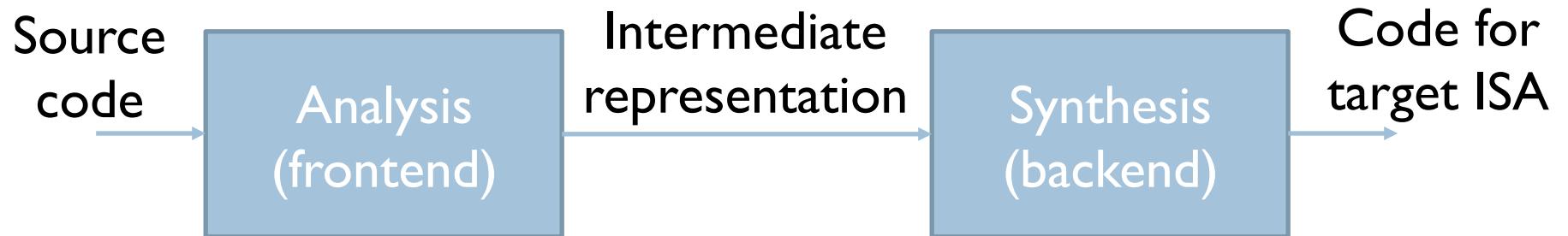
```
r = 1;
start:
    CMOVE(1, r0)
    ST(r0, r)
    LD(n,r1)    | keep n in r1
    LD(r,r3)    | keep r in r3
```

```
while (n > 0)
{
    r = r*n;
    n = n-1;
}
loop:
    BR(test)
test:
    MUL(r1, r3, r3)
    SUBC(r1, 1, r1)
    CMPLT(r31, r1, r2)
    BT(r2, loop)
```

Optimization:
Keep n, r in registers
⇒ move LDs/STs
out of loop!
4 instructions in the loop

```
done:
    ST(r1,n)    | save final n
    ST(r3,r)    | save final r
```

Anatomy of a Modern Compiler

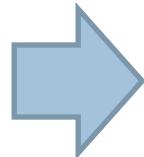


- Read source program
- Break it up into basic elements
- Check correctness, report errors
- Translate to generic **intermediate representation (IR)**
- Optimize IR
- Translate IR to ASM
- Optimize ASM

Frontend Stages

- Lexical analysis (scanning): Source → List of tokens

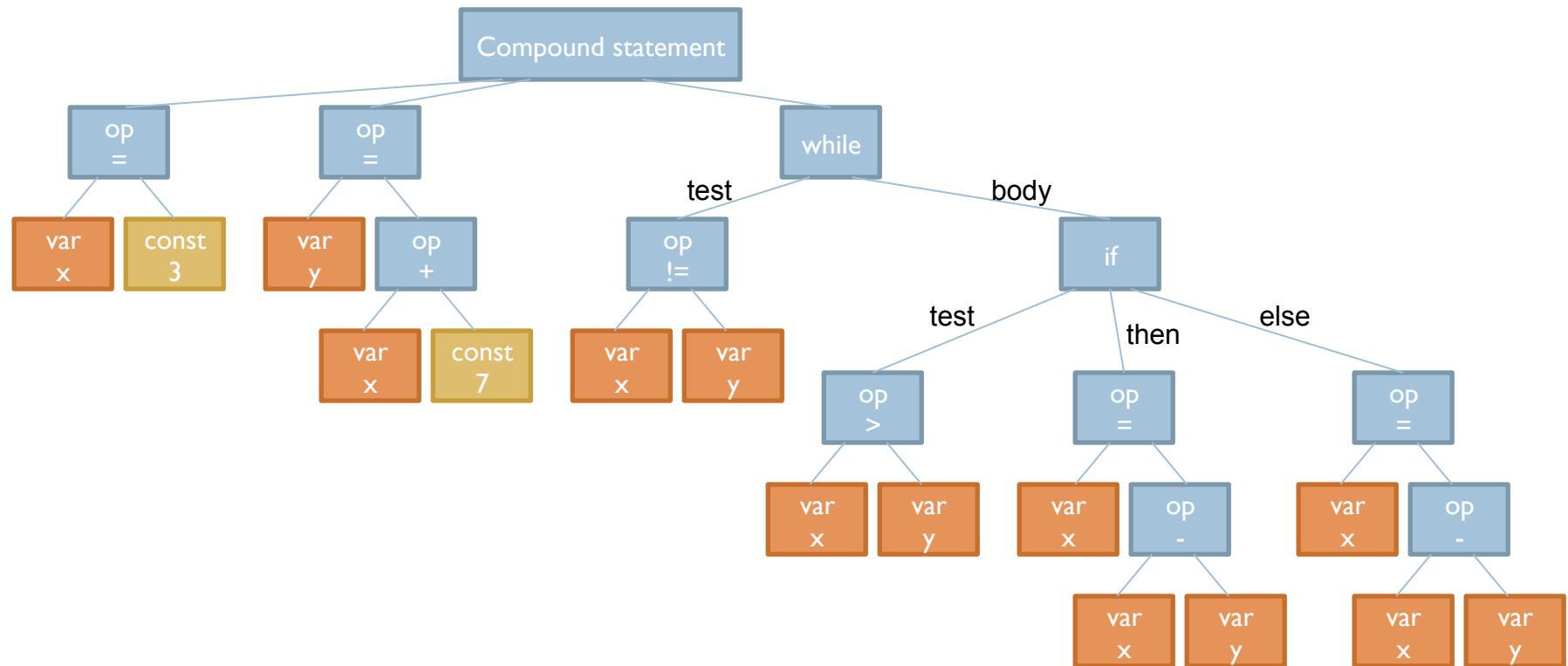
```
int x = 3;  
int y = x + 7;  
while (x != y) {  
    if (x > y) {  
        x = x - y;  
    } else {  
        y = y - x;  
    }  
}
```



```
(“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)  
 ...
```

Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree



Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree
- Semantic analysis (mainly, type checking)

Consider:

`int x = "bananas";`

Syntax OK

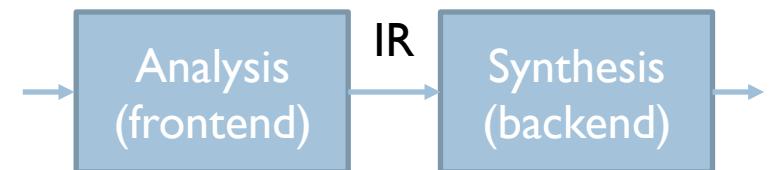
Semantically (meaning)
WRONG



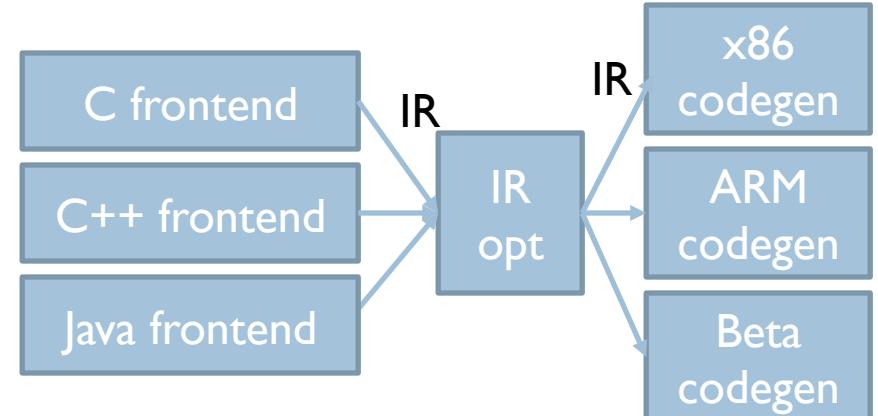
Line 1: error, invalid conversion from string constant to int

Intermediate Representation (IR)

- Internal compiler language that is:
 - Language-independent
 - Machine-independent
 - Easy to optimize

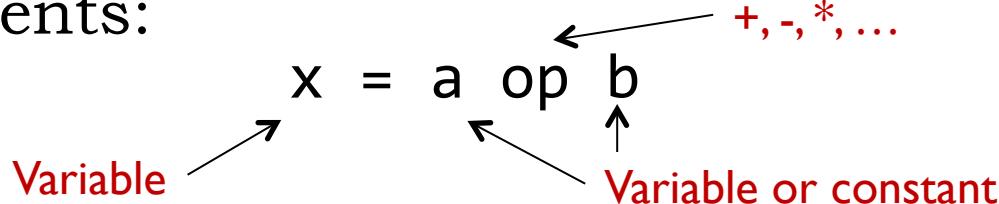


- Why yet another language?
 - Assembly does not have enough info to optimize it well
 - Enables modularity and reuse



Common IR: Control Flow Graph

- Assignments:



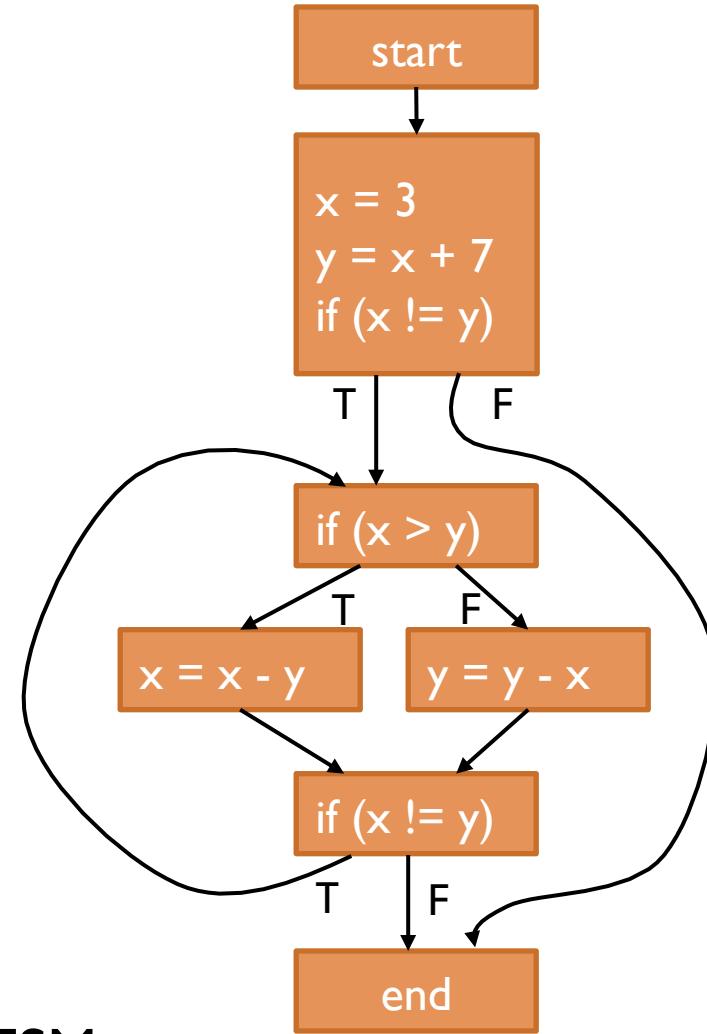
- Basic block: Sequence of assignments with an optional branch at the end

```
x = 3  
y = x + 7  
if (x != y)
```

- Control flow graph:
 - Nodes: Basic blocks
 - Edges: branches between basic blocks

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;  
    }  
}
```



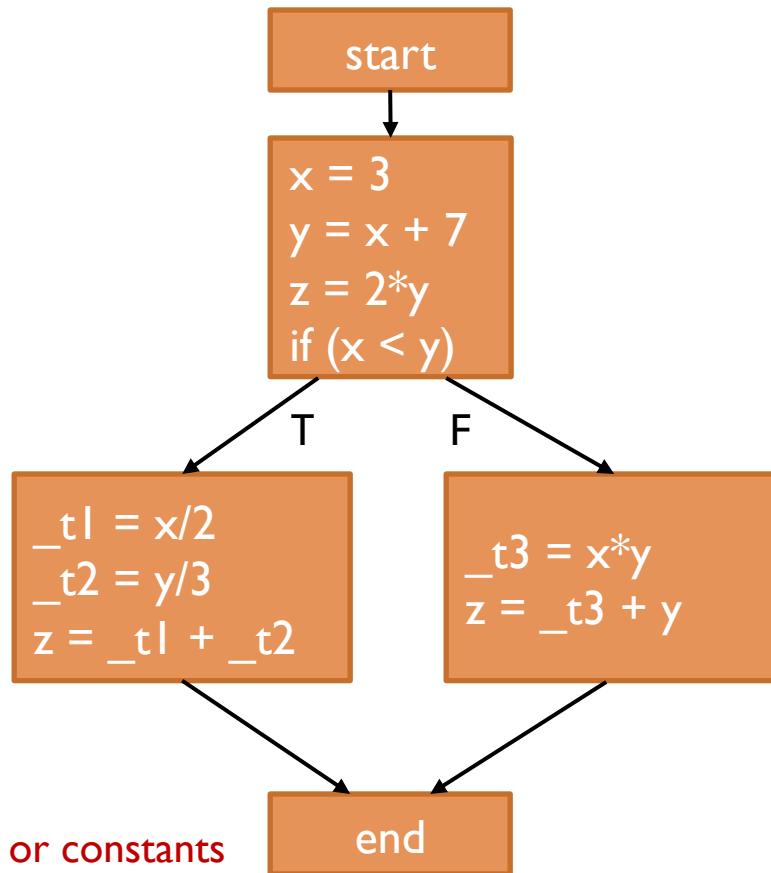
Looks like a high-level FSM...

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

Example IR Optimizations

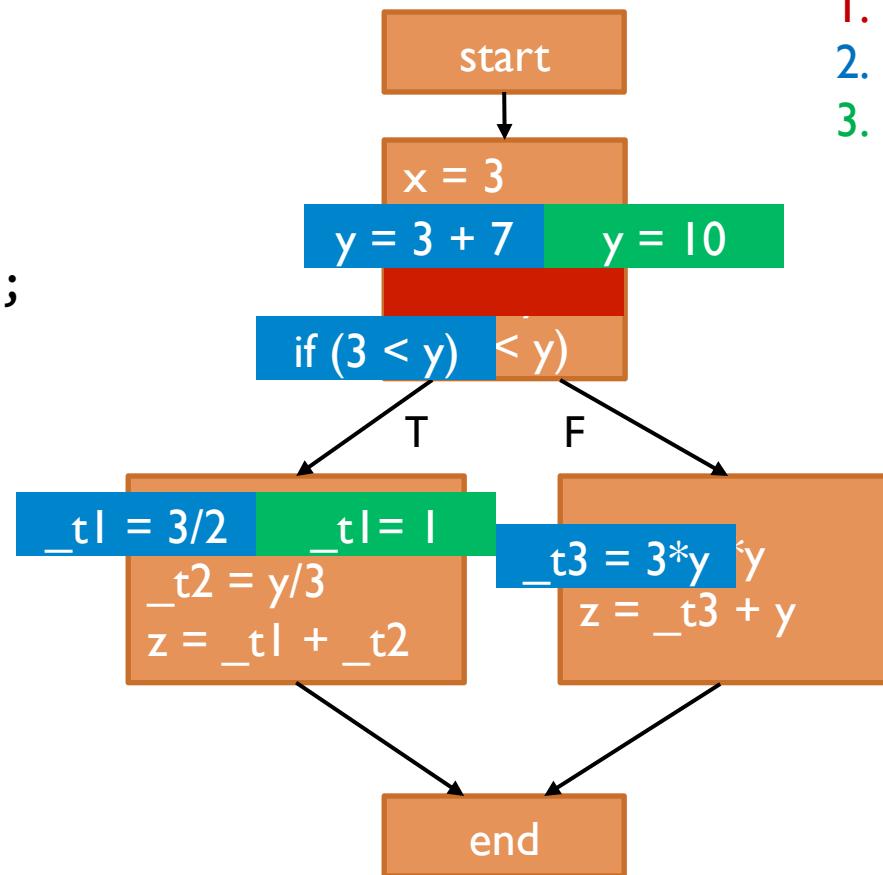
```
int x = 3;  
int y = x + 7;  
int z = 2*y;  
if (x < y) {  
    z = x/2 + y/3;  
} else {  
    z = x*y + y;  
}
```



NOTE: Expressions with > 2 vars or constants
broken down in multiple assignments,
using temporary variables

Example IR Optimizations

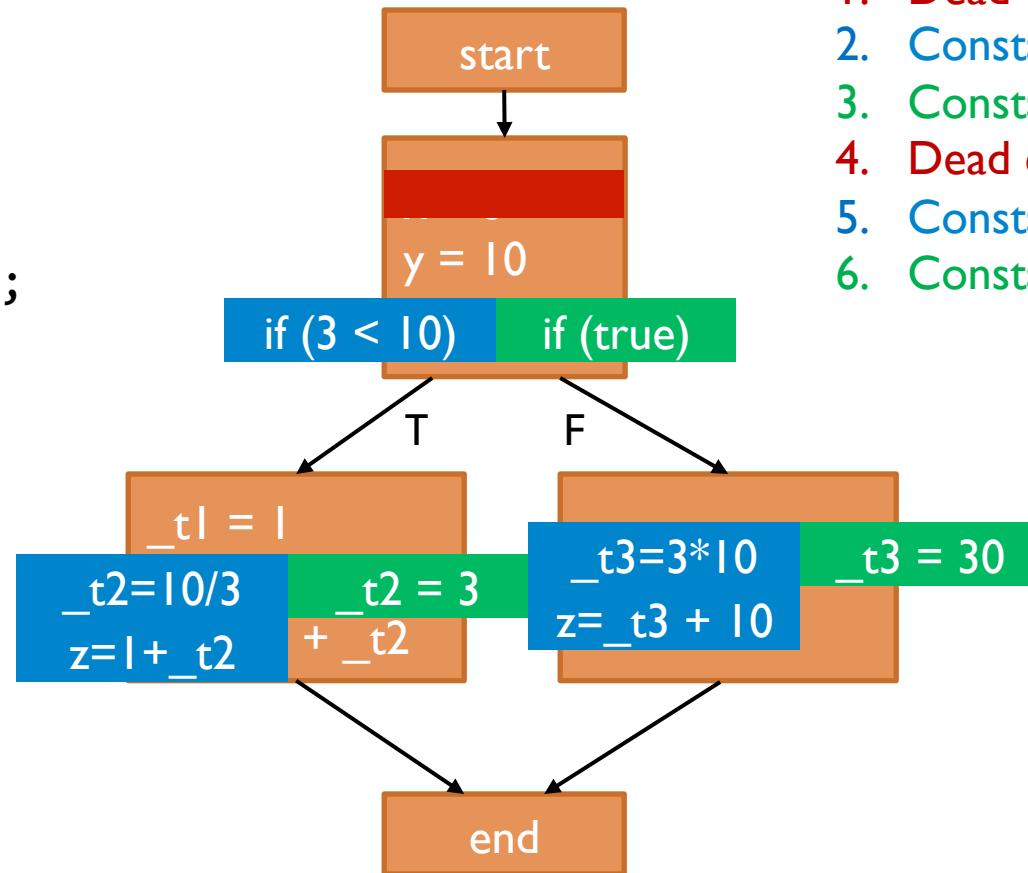
```
int x = 3;  
int y = x + 7;  
int z = 2*y;  
if (x < y) {  
    z = x/2 + y/3;  
} else {  
    z = x*y + y;  
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding

Example IR Optimizations

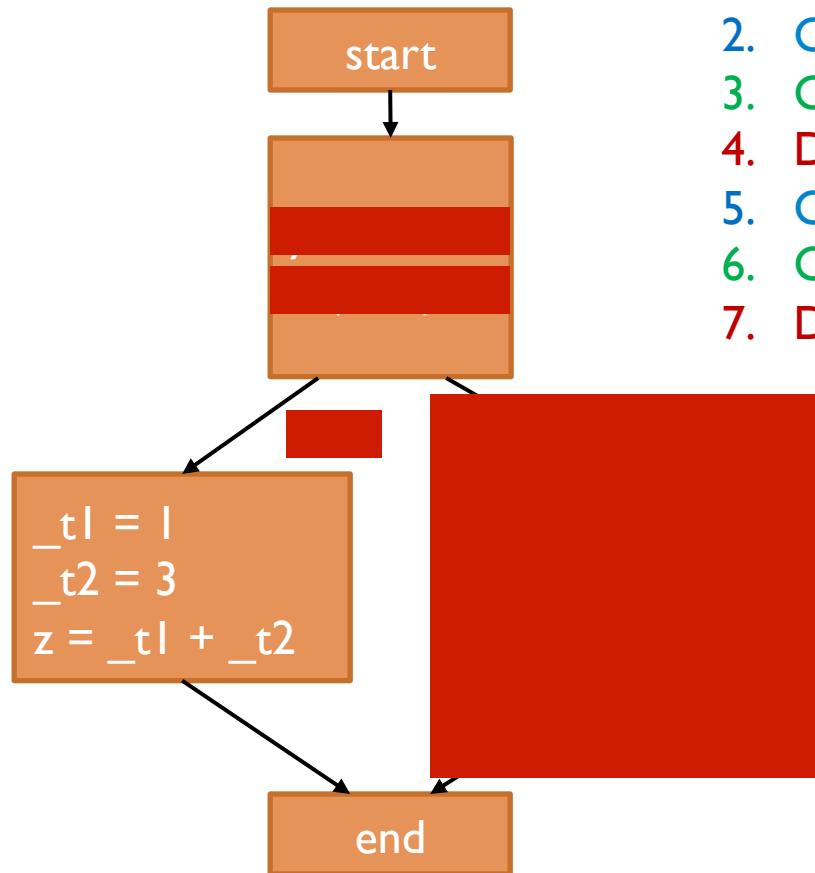
```
int x = 3;  
int y = x + 7;  
int z = 2*y;  
if (x < y) {  
    z = x/2 + y/3;  
} else {  
    z = x*y + y;  
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding

Example IR Optimizations

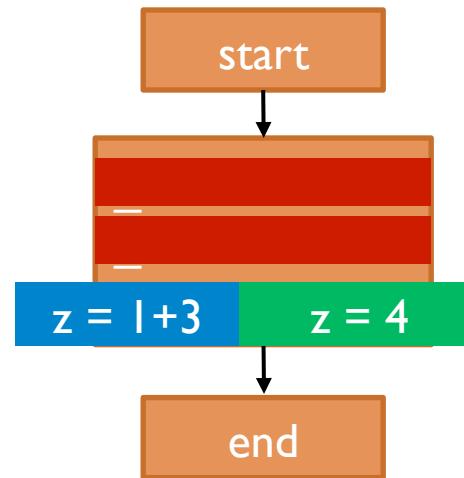
```
int x = 3;  
int y = x + 7;  
int z = 2*y;  
if (x < y) {  
    z = x/2 + y/3;  
} else {  
    z = x*y + y;  
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim

Example IR Optimizations

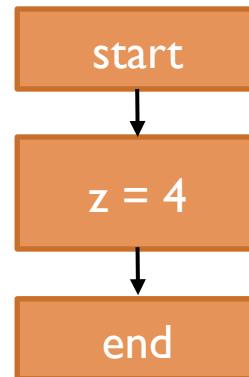
```
int x = 3;  
int y = x + 7;  
int z = 2*y;  
if (x < y) {  
    z = x/2 + y/3;  
} else {  
    z = x*y + y;  
}
```



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

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

Code Generation

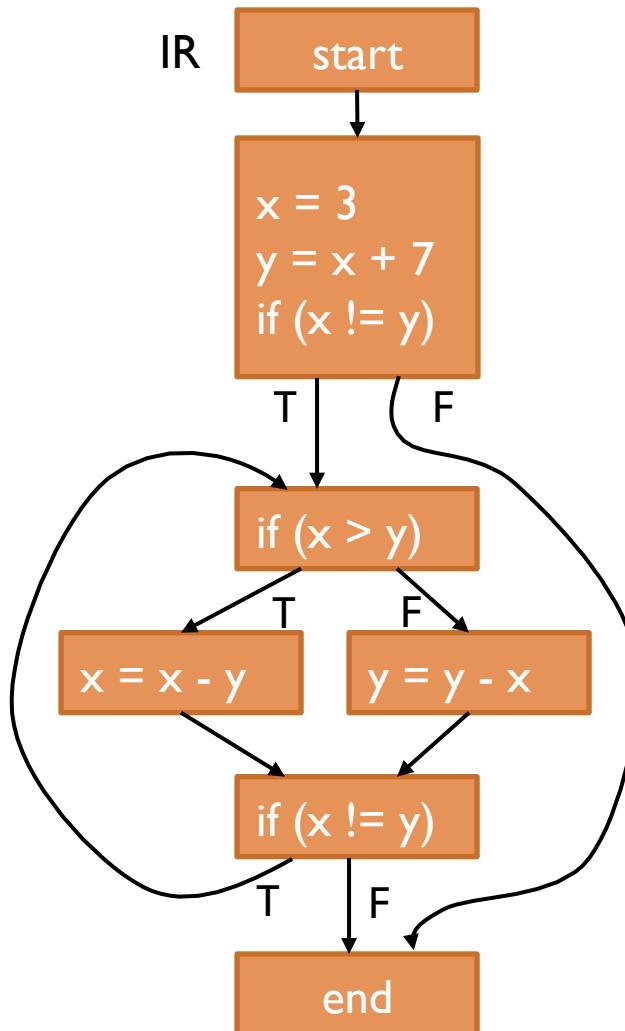
- Translate generated IR to assembly
- Register allocation: Map variables to registers
 - If variables > registers, map some to memory, and load/store them when needed
- Translate each assignment to instructions
 - Some assignments may require > 1 instr if our ISA doesn't have op
- Emit each basic block: 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

Putting It All Together: GCD

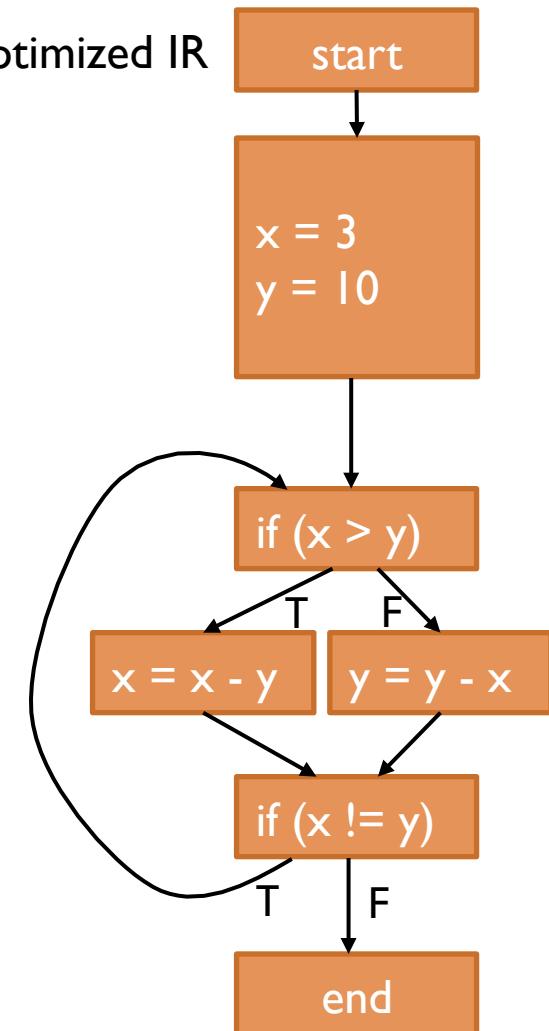
Source code

```
int x = 3;  
int y = x + 7;  
while (x != y) {  
    if (x > y) {  
        x = x - y;  
    } else {  
        y = y - x;  
    }  
}
```

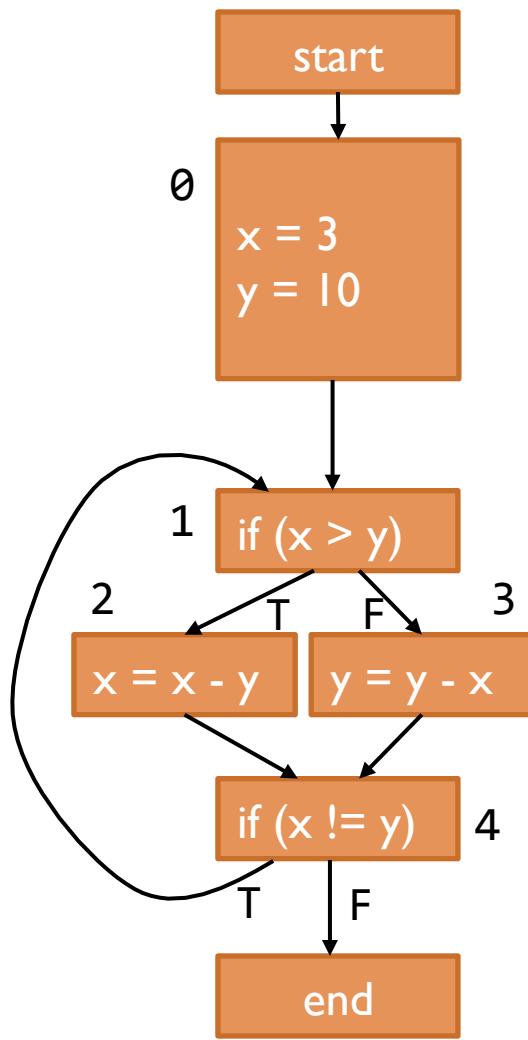
IR



Optimized IR



Putting It All Together: GCD



1. Allocate registers:

$x: R0, y: R1$

2. Produce each basic block:

BBL0: CMOVE(3, R0)
CMOVE(10, R1)
BR(BBL1)

BBL1: CMPLT(R1, R0, R2)
BT(R2, BBL2)
BR(BBL3)

BBL2: SUB(R0, R1, R0)
BR(BBL4)

BBL3: SUB(R1, R0, R1)
BR(BBL4)

BBL4: CMPEQ(R1, R0, R2)
BT(R2, end)
BR(BBL1)

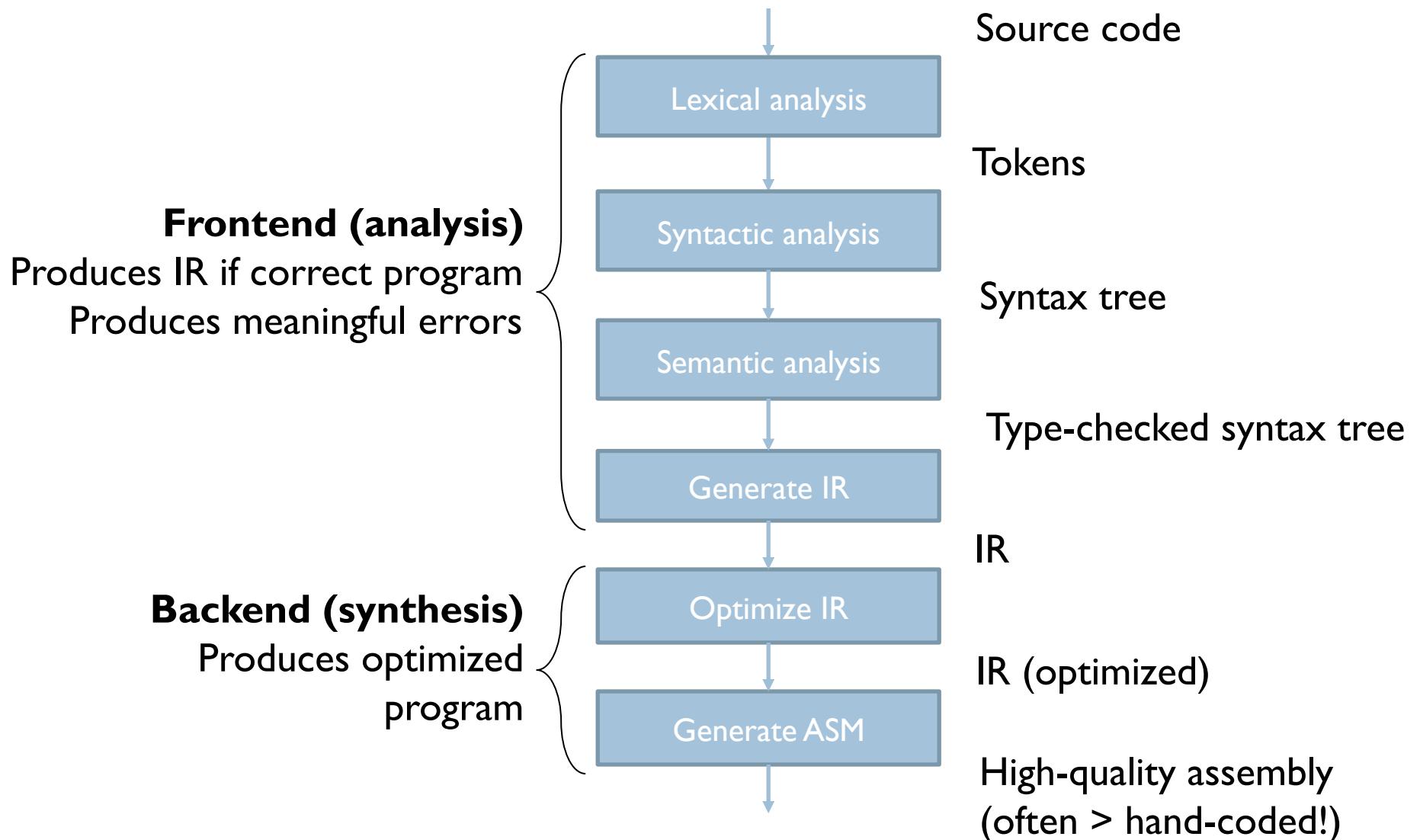
end:

3. Lay out BBs, removing superfluous branches:

BBL0:	CMOVE(3, R0)
	CMOVE(10, R1)
BBL1:	CMPLT(R1, R0, R2)
	BT(R2, BBL2)
BBL3:	SUB(R1, R0, R1)
	BR(BBL4)
BBL2:	SUB(R0, R1, R0)
BBL4:	CMPEQ(R1, R0, R2)
	BF(R2, BBL1)

end:

Summary: Modern Compilers



12. Procedures & Stacks

6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2015 MIT EECS

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
```

Implementing Procedures

- Option 1: Inlining
 - Compiler substitutes procedure call with body
 - Problems?
 - Code size
 - Recursion

```
int fact(int n) {  
    if (n > 0) {  
        return n*fact(n - 1);  
    } else {  
        return 1;  
    }  
}
```

- 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

Procedure Linkage: First Try

```
int fact(int n) {  
    if (n > 0) {  
        return n*fact(n - 1);  
    } else {  
        return 1;  
    }  
}
```

```
fact(3);
```

```
fact(3) = 3*fact(2)  
fact(2) = 2*fact(1)  
fact(1) = 1*fact(0)  
fact(0) = 1
```

- 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

Procedure Linkage: First Try

```
int fact(int n) {  
    if (n > 0) {  
        return n*fact(n - 1);  
    } else {  
        return 1;  
    }  
}  
  
fact(3);
```

fact:

```
CMPLEC(r1,0,r0)  
BT(r0,else)  
MOVE(r1,r2) // save n  
SUBC(r2,1,r1)  
BR(fact,r28) OOPS!  
MUL(r0,r2,r0)  
BR(rtn)  
  
else: CMOVE(1,r0)  
rtn: JMP(r28)
```

```
main: CMOVE(3,r1)  
      BR(fact,r28)  
      HALT()
```

- Proposed convention:
 - Pass argument (value of n) in R1
 - Pass return address in R28
 - Return result in R0

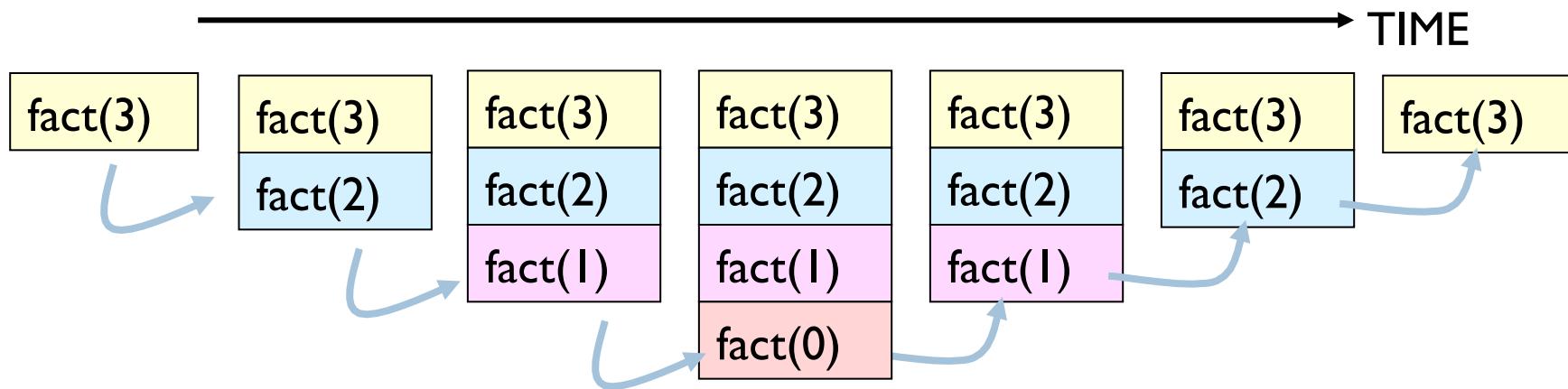
Procedure Storage Needs

- 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*

Activation Records

```
int fact(int n) {  
    if (n > 0) return n*fact(n - 1);  
    else return 1;  
}
```

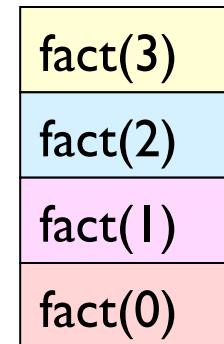


A procedure call creates a new activation record. Caller's record is preserved because we'll need it when callee finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

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



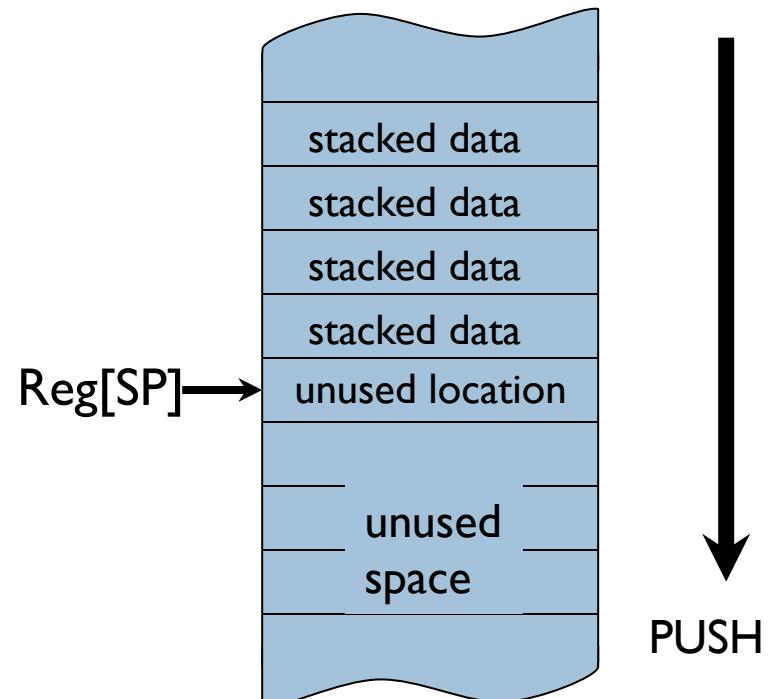
- Stack: push, pop, access to top element
- For C, we only need to access to the activation record of the currently executing procedure

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

LOWER ADDRESSES



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.

Stack Management Macros

PUSH (RX): Push Reg[x] onto stack

$$\text{Reg[SP]} \leftarrow \text{Reg[SP]} + 4;$$

$$\text{Mem}[\text{Reg[SP]}-4] \leftarrow \text{Reg}[x]$$

ADD(R29, 4, R29)
ST(RX, -4, R29)

POP (RX): Pop value on top of the stack into Reg[x]

$$\text{Reg}[x] \leftarrow \text{Mem}[\text{Reg[SP]}-4]$$

$$\text{Reg[SP]} \leftarrow \text{Reg[SP]} - 4;$$

LD(R29, -4, RX)
SUBC(R29, 4, R29)

ALLOCATE (k): Reserve k WORDS of stack

$$\text{Reg[SP]} \leftarrow \text{Reg[SP]} + 4*k$$

ADD(R29, 4*k, R29)

DEALLOCATE (k): Release k WORDS of stack

$$\text{Reg[SP]} \leftarrow \text{Reg[SP]} - 4*k$$

SUBC(R29, 4*k, R29)

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(dilithum_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..

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(expr_n) $\Rightarrow Rx$

PUSH(rx)

...

compile_expr(expr₁) $\Rightarrow Rx$

PUSH(rx)

BR(proc, LP)

DEALLOCATE(n)

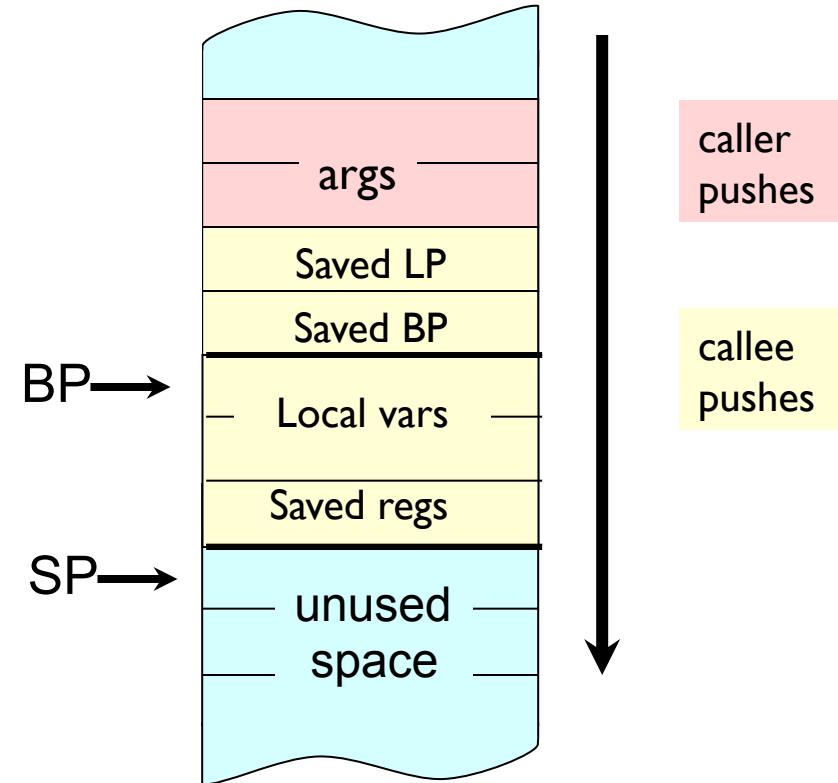
Stack Frames as Activation Records

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 PUSHs and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

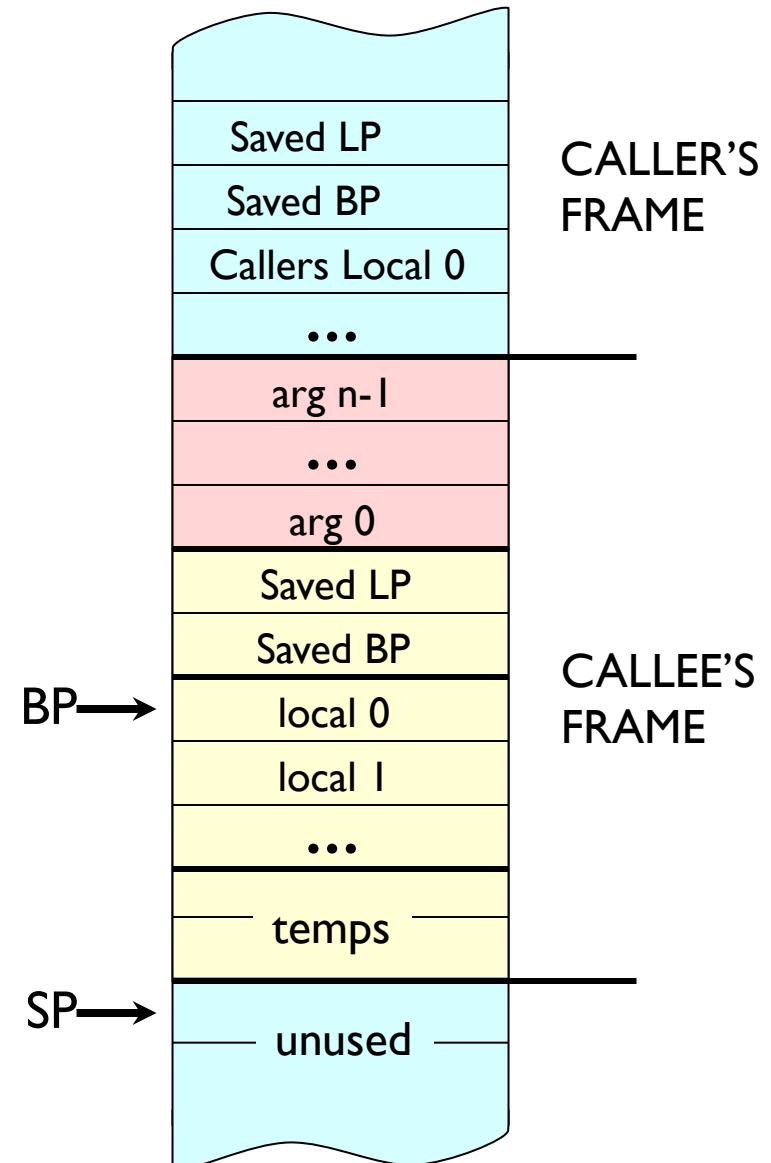
Stack Frame Details

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?



Argument Order & BP usage

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 ($j \geq 0$):

$\text{LD}(\text{BP}, -4*(j+3), \text{rx})$

or

$\text{ST}(\text{rx}, -4*(j+3), \text{BP})$

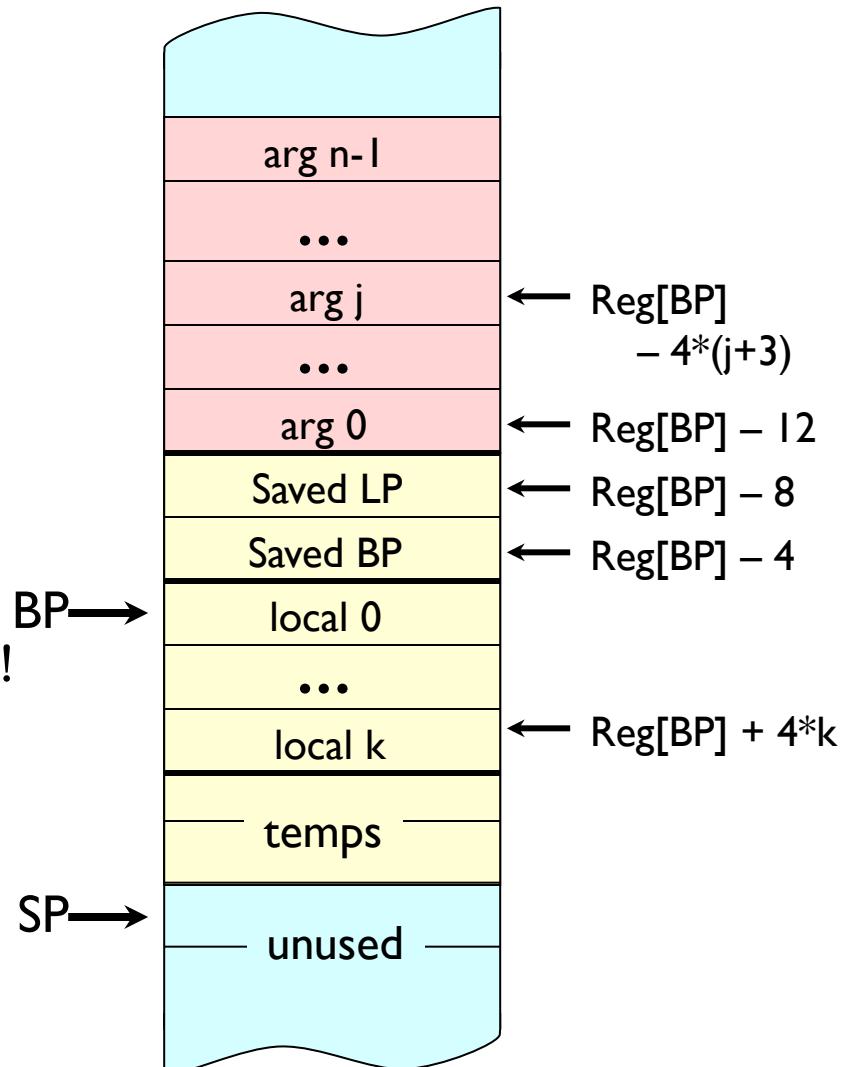
CALLEE can access the first few arguments without knowing how many arguments have been passed!

- 2) To access k^{th} local variable ($k \geq 0$)

$\text{LD}(\text{BP}, k*4, \text{rx})$

or

$\text{ST}(\text{rx}, k*4, \text{BP})$



Procedure Linkage: The Contract

The CALLER will:

- Push args onto stack, in reverse order.
- Branch to callee, putting return address into LP.
- Remove args from stack on return.

The CALLEE will:

- Perform promised computation, leaving result in R0.
- Branch to return address.
- Leave stacked data intact, including stacked args.
- Leave regs (except R0) unchanged.

Procedure Linkage

typical “boilerplate” templates

Calling Sequence

```
PUSH(argn)          // push args, last arg first  
...  
PUSH(arg1)  
BR(f, LP)           // Call f.  
DEALLOCATE(n)       // Clean up!  
...                 // (f's return value in r0)
```

Entry Sequence

```
f: PUSH(LP)          // Save LP and BP  
PUSH(BP)           // in case we make new calls.  
MOVE(SP,BP)         // set BP=frame base  
ALLOCATE(nlocals)  // allocate locals  
(push other regs) // 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.
```

Putting It All Together: Factorial

```
int fact(int n) {
    if (n > 0) {
        return n*fact(n-1);
    } else {
        return 1;
    }
}
```

	fact:	PUSH(LP) // save linkages PUSH(BP) MOVE(SP,BP) PUSH(r1) // preserve regs LD(BP,-12,r1) // r1 ← n CMPLEC(r1,0,r0) // if (n > 0) BT(r0,else)
		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) POP(BP) POP(LP) JMP(LP) // return

Recursion?

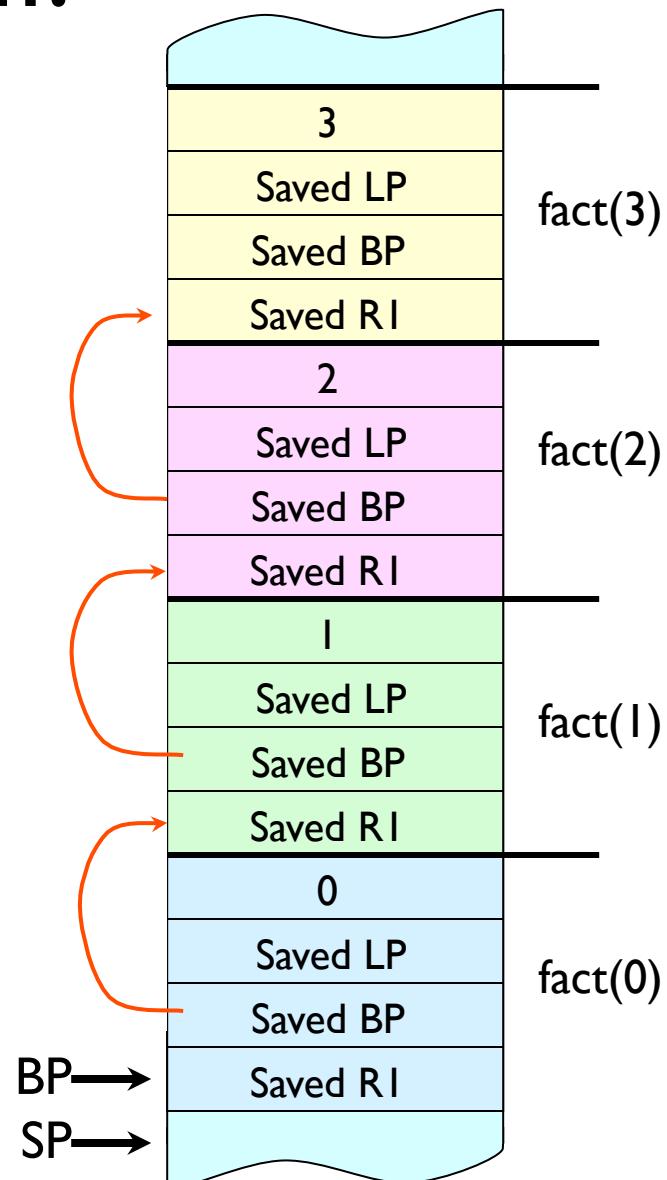
Of course!

- 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

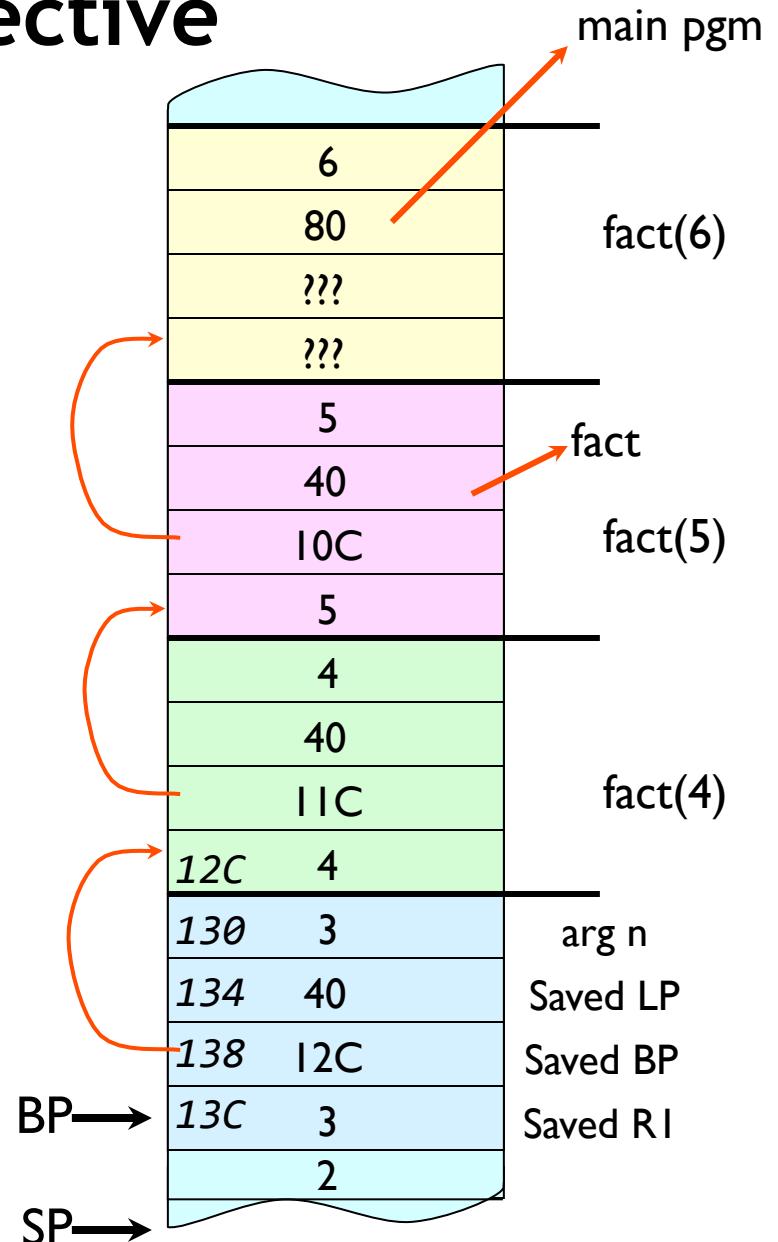
Particularly useful on 6.004 quizzes!



Stack Detective

`fact(n)` is called. During the calculation, the computer is stopped with the PC at `0x40`; the stack contents are shown (in hex).

- What's the argument to the *active* call to `fact`? **3**
- What's the argument to the *original* call to `fact`? **6**
- What's the location of the original calling (BR) instruction? **$80 - 4 = 7C$**
- What instruction is about to be executed? **DEALLOCATE(1)**
- What value is in BP? **13C**
- What value is in SP? **$13C + 4 + 4 = 144$**
- What value is in R0? **`fact(2) = 2`**



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]

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

13. Building the Beta

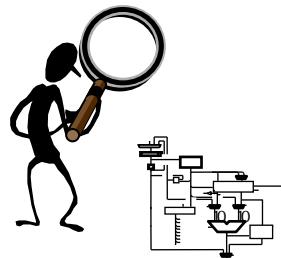
6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2015 MIT EECS

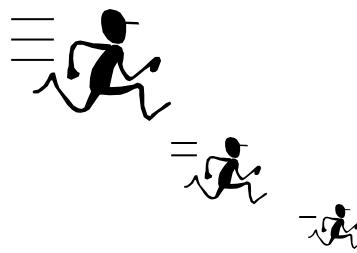
CPU Design Tradeoffs



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.

Processor Performance

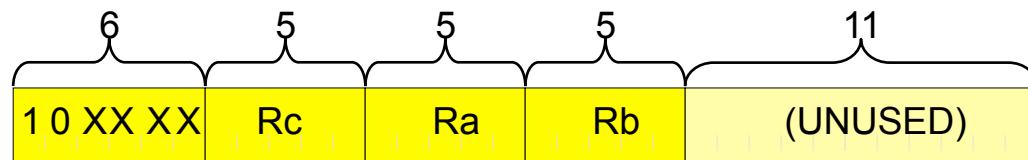
- “Iron Law” of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

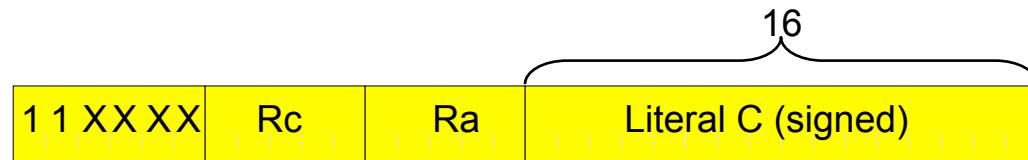
$$\text{Perf} = \frac{1}{\text{Time}}$$

- 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

Reminder: Beta ISA



Operate class: $Reg[R_c] \leftarrow Reg[R_a] \text{ op } Reg[R_b]$



Operate class: $Reg[R_c] \leftarrow Reg[R_a] \text{ op } SXT(C)$

Opcodes, both formats:

ADD	SUB	MUL*	DIV*	*optional
CMPEQ	CMPLE	CMPLT		
AND	OR	XOR		
SHL	SHR	SRA		



LD: $Reg[R_c] \leftarrow Mem[Reg[R_a]+SXT(C)]$

ST: $Mem[Reg[R_a]+SXT(C)] \leftarrow Reg[R_c]$

LDR: $Reg[R_c] \leftarrow Mem[PC + 4 + 4*SXT(C)]$

BEQ: $Reg[R_c] \leftarrow PC+4; \text{ if } Reg[R_a]=0 \text{ then } PC \leftarrow PC+4+4*SXT(C)$

BNE: $Reg[R_c] \leftarrow PC+4; \text{ if } Reg[R_a]\neq0 \text{ then } PC \leftarrow PC+4+4*SXT(C)$

JMP: $Reg[R_c] \leftarrow PC+4; \text{ PC } \leftarrow Reg[R_a]$

Instruction classes distinguished by OPCODE:

OP

OPC

MEM

Control Flow

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:



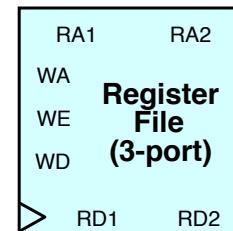
Registers



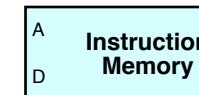
Muxes



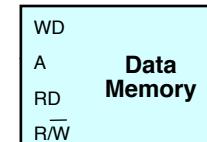
“Black box” ALU



Registers

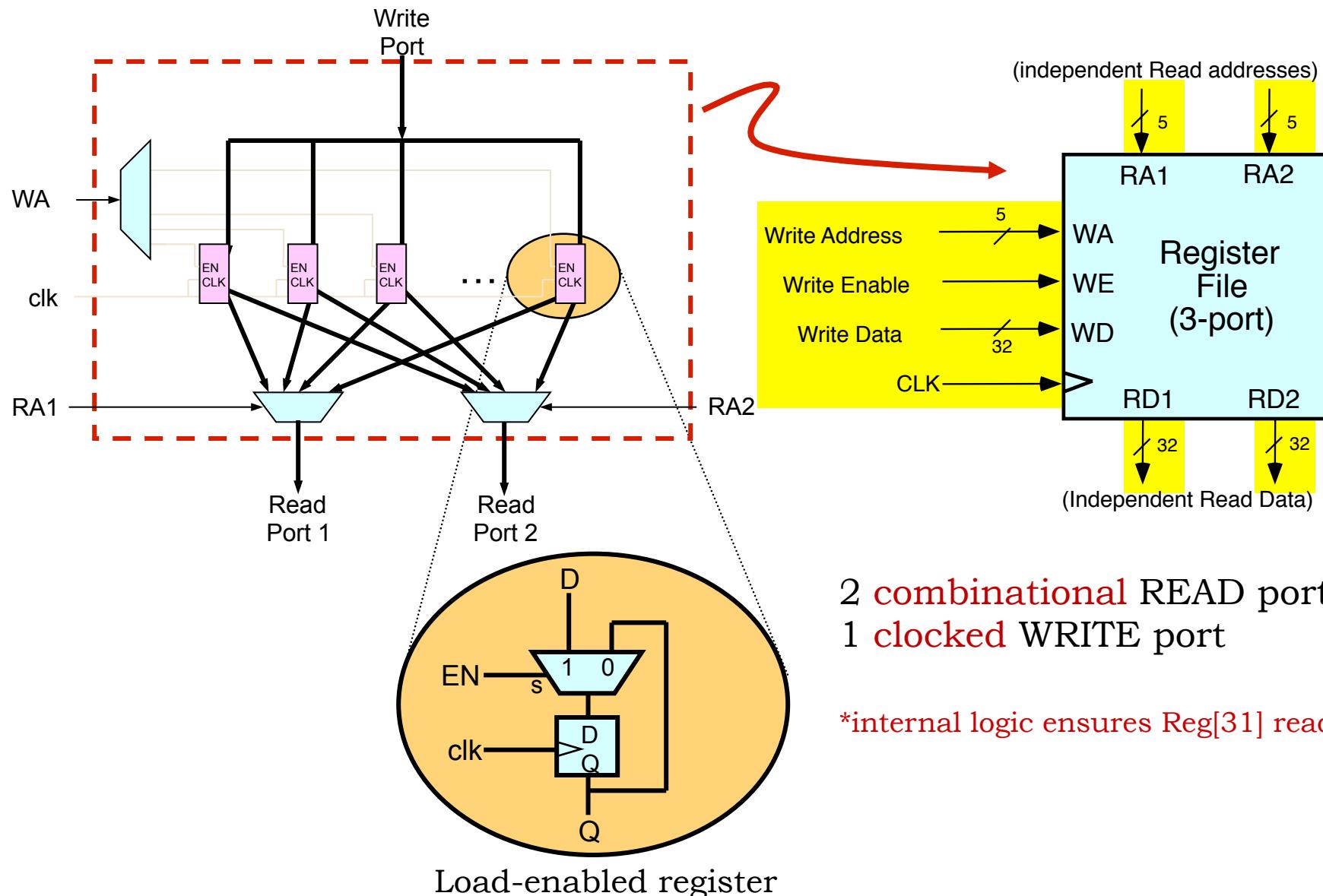


Instruction Memory



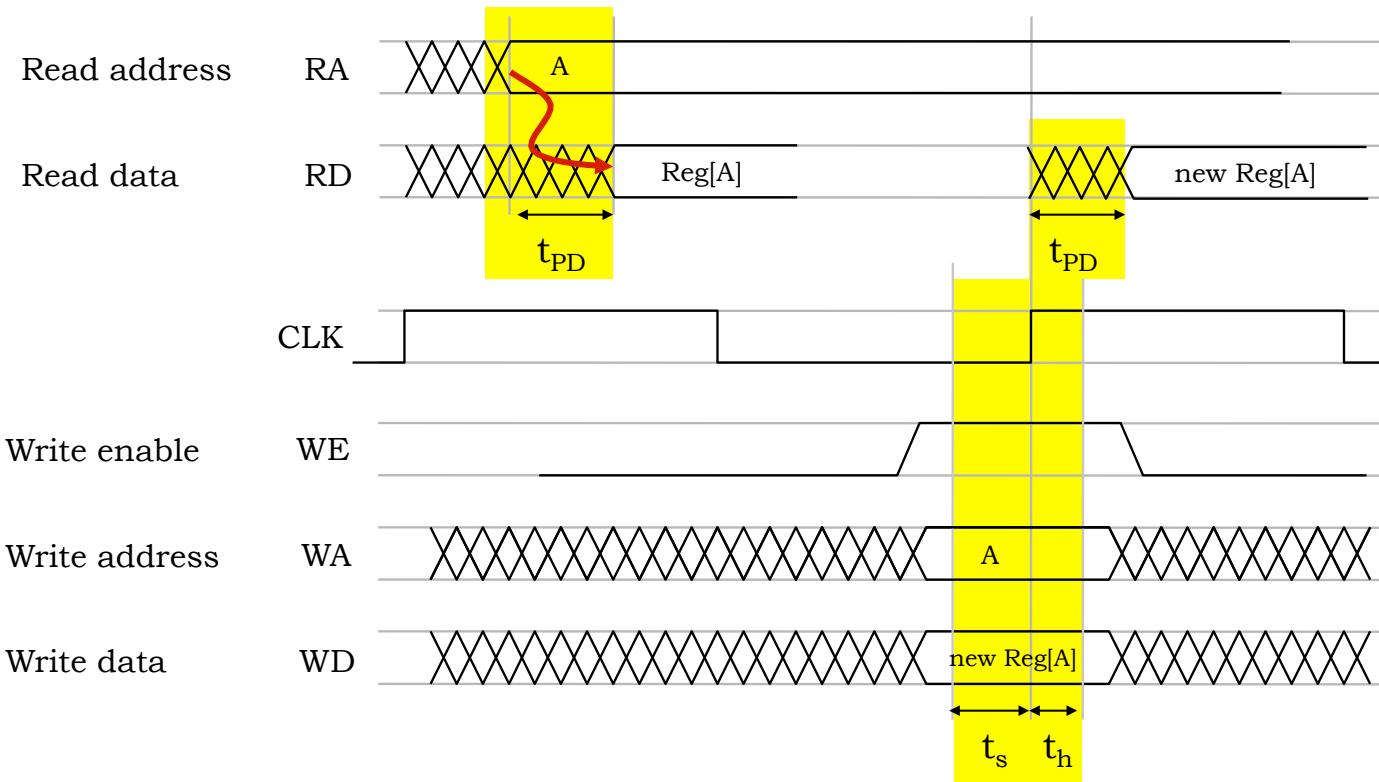
Data Memory

Multi-Ported Register File



Register File Timing

2 **combinational** READ ports, 1 **clocked** WRITE port

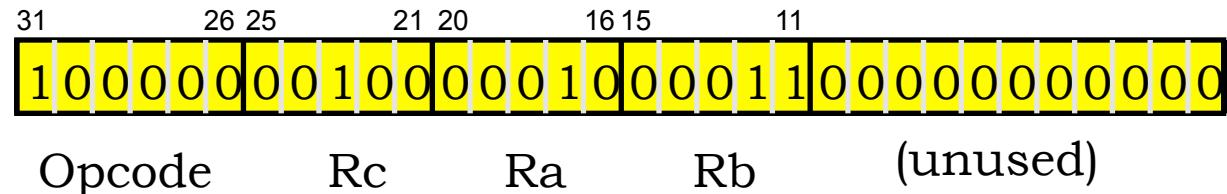


What if (say) $WA=RA1???$

RD1 reads “old” value of $\text{Reg}[RA1]$ until next clock edge!

ALU Instructions

32-bit (4-byte) ADD instruction:



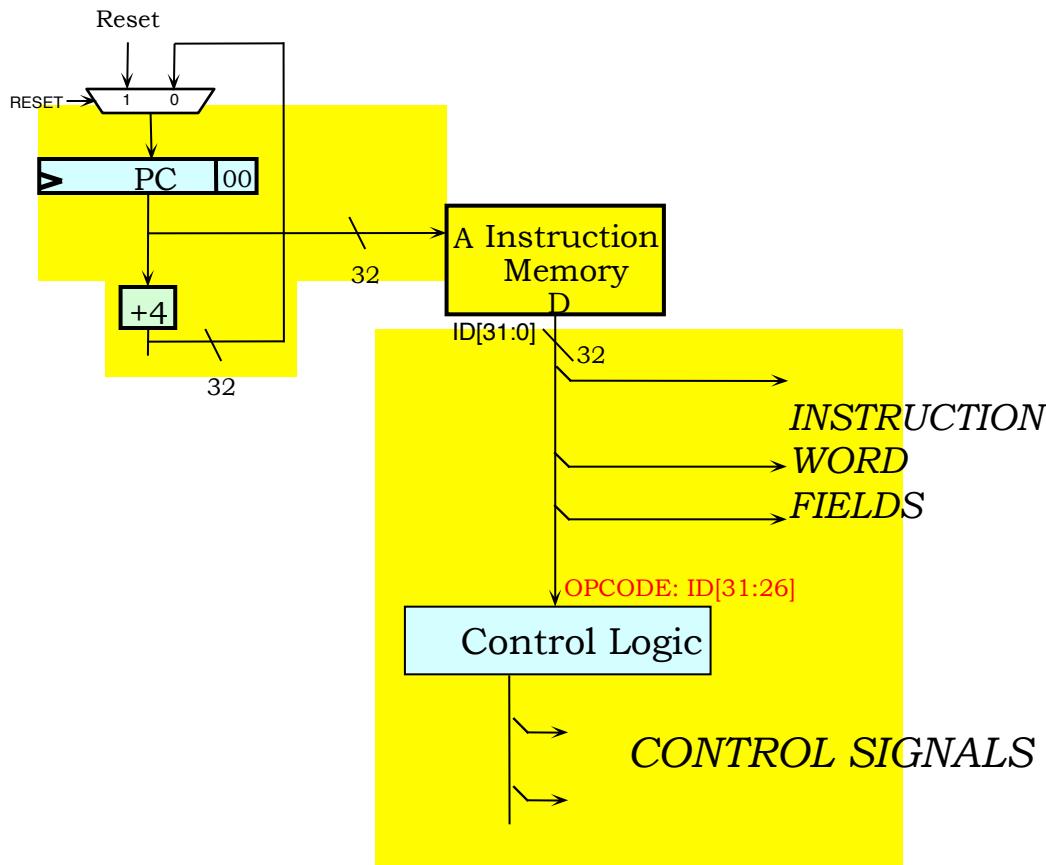
Means, to Beta, $Reg[R4] \leftarrow Reg[R2] + Reg[R3]$

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)

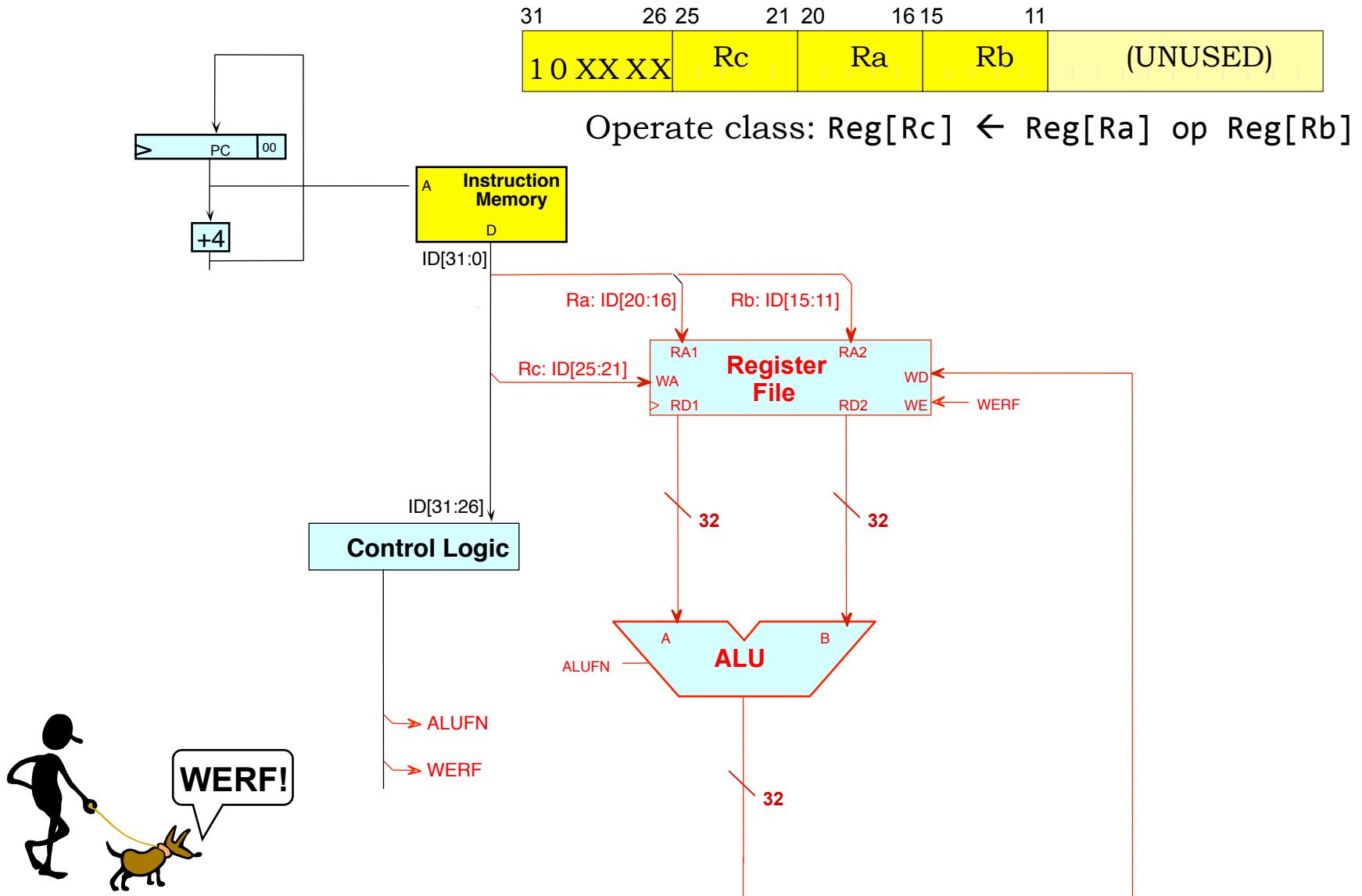
Instruction Fetch/Decode

Use a counter to FETCH the next instruction: PROGRAM COUNTER (PC)

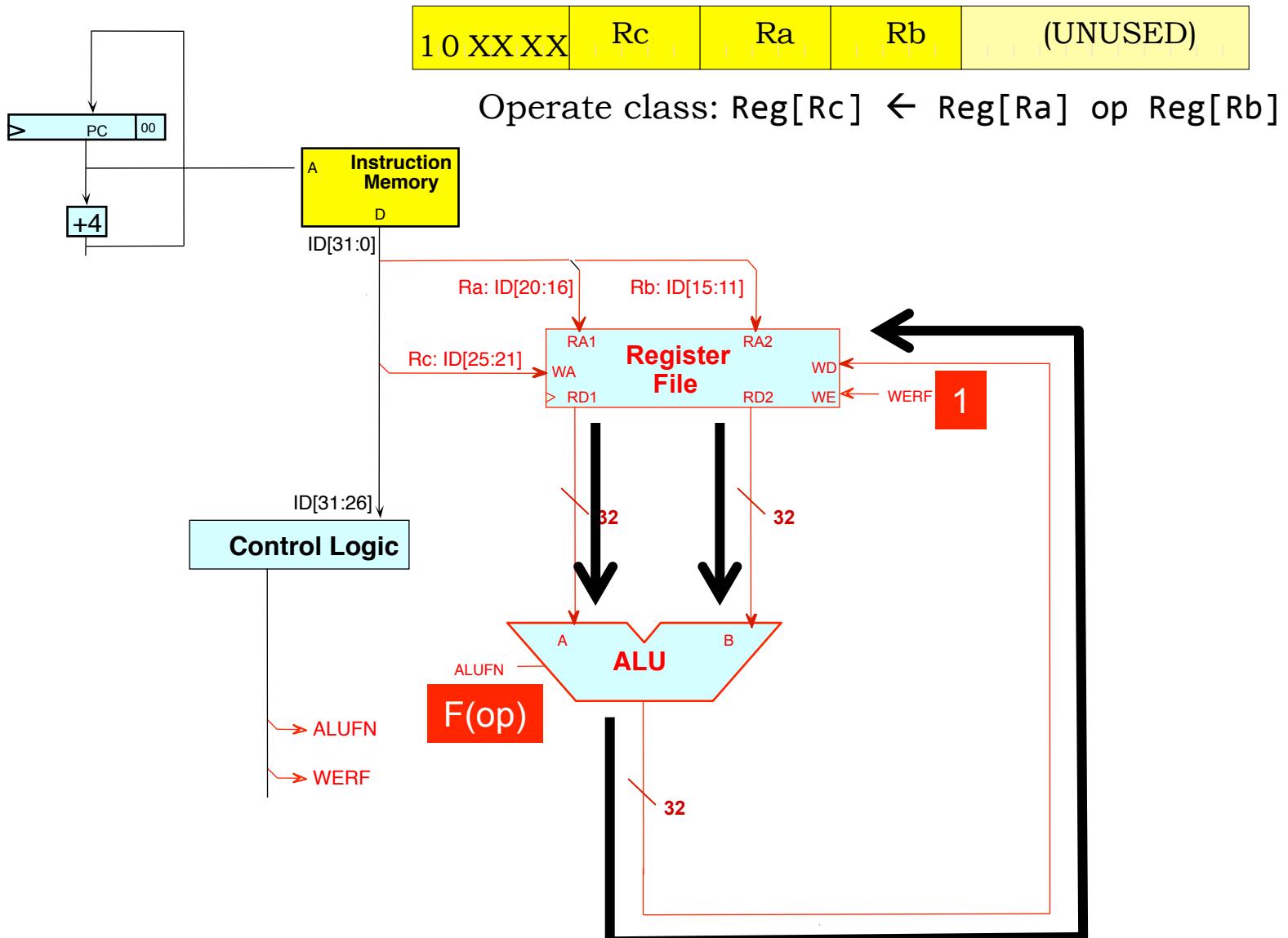


- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory
 - Use some instruction fields directly (register numbers, 16-bit constant)
- Use bits [31:26] to generate control signals

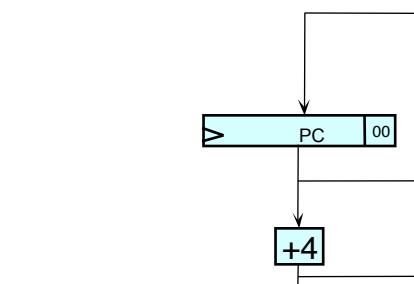
ALU Op Datapath



ALU Op Datapath



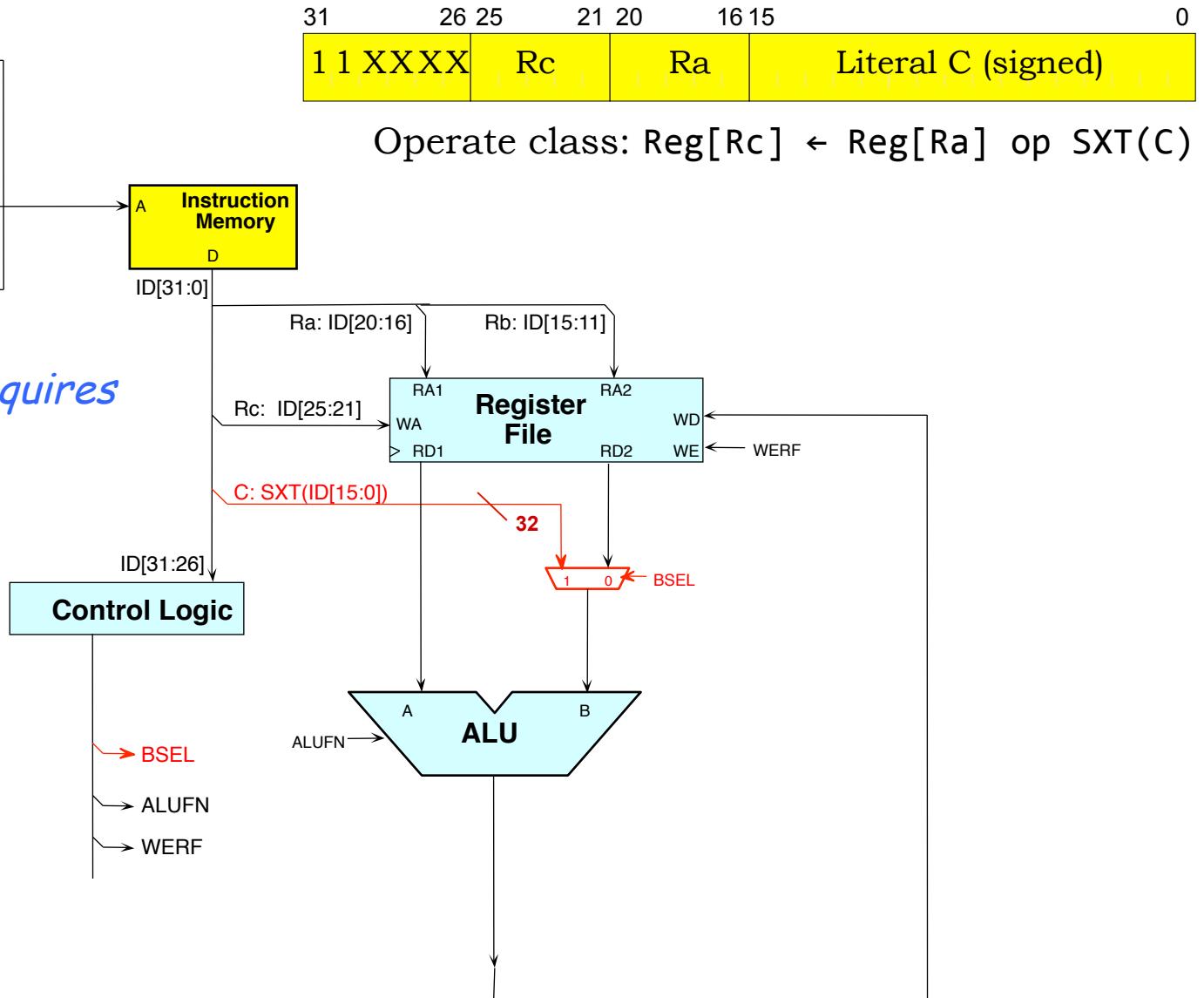
ALU Operations (with constant)



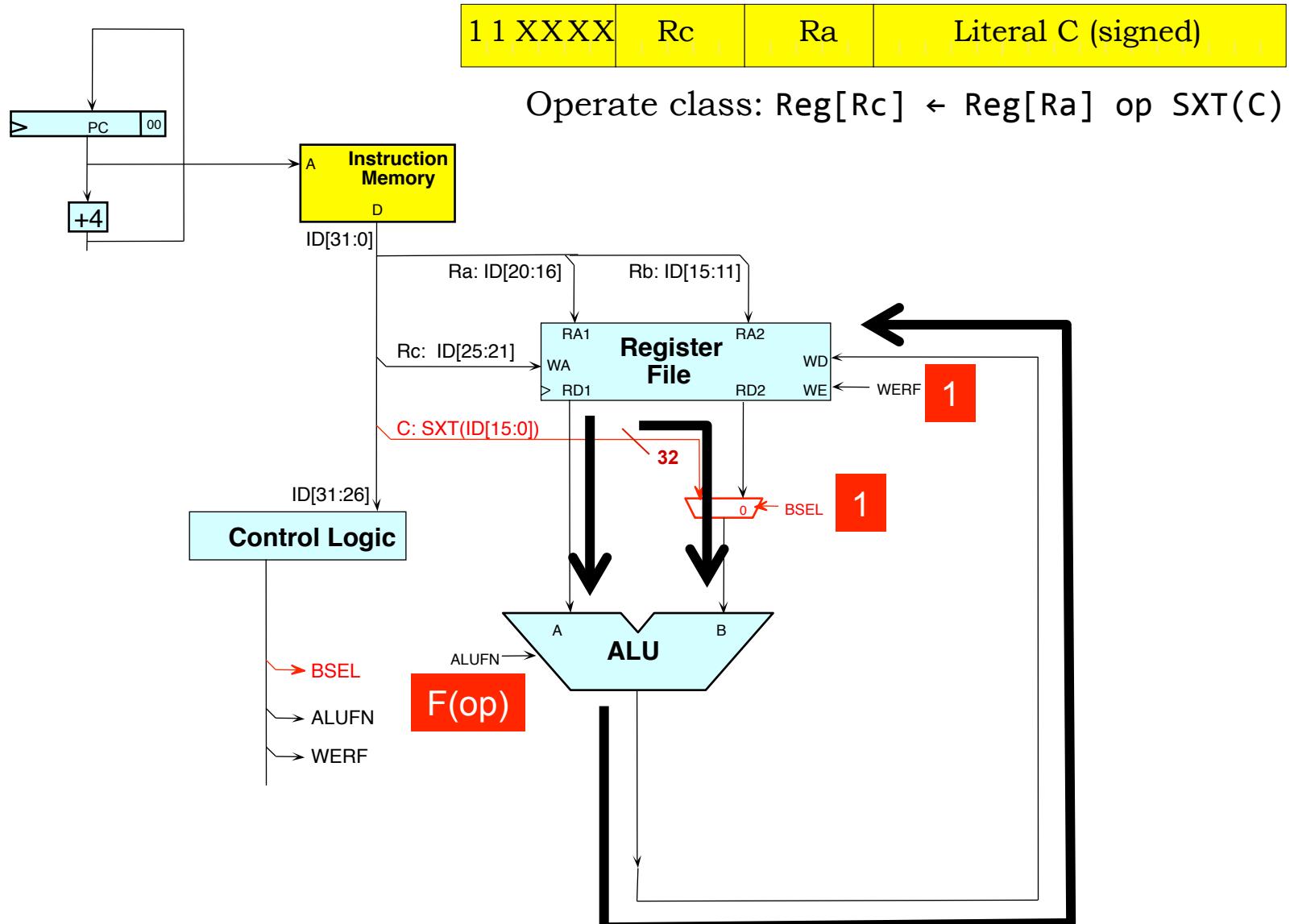
Sign-extension requires no logic...



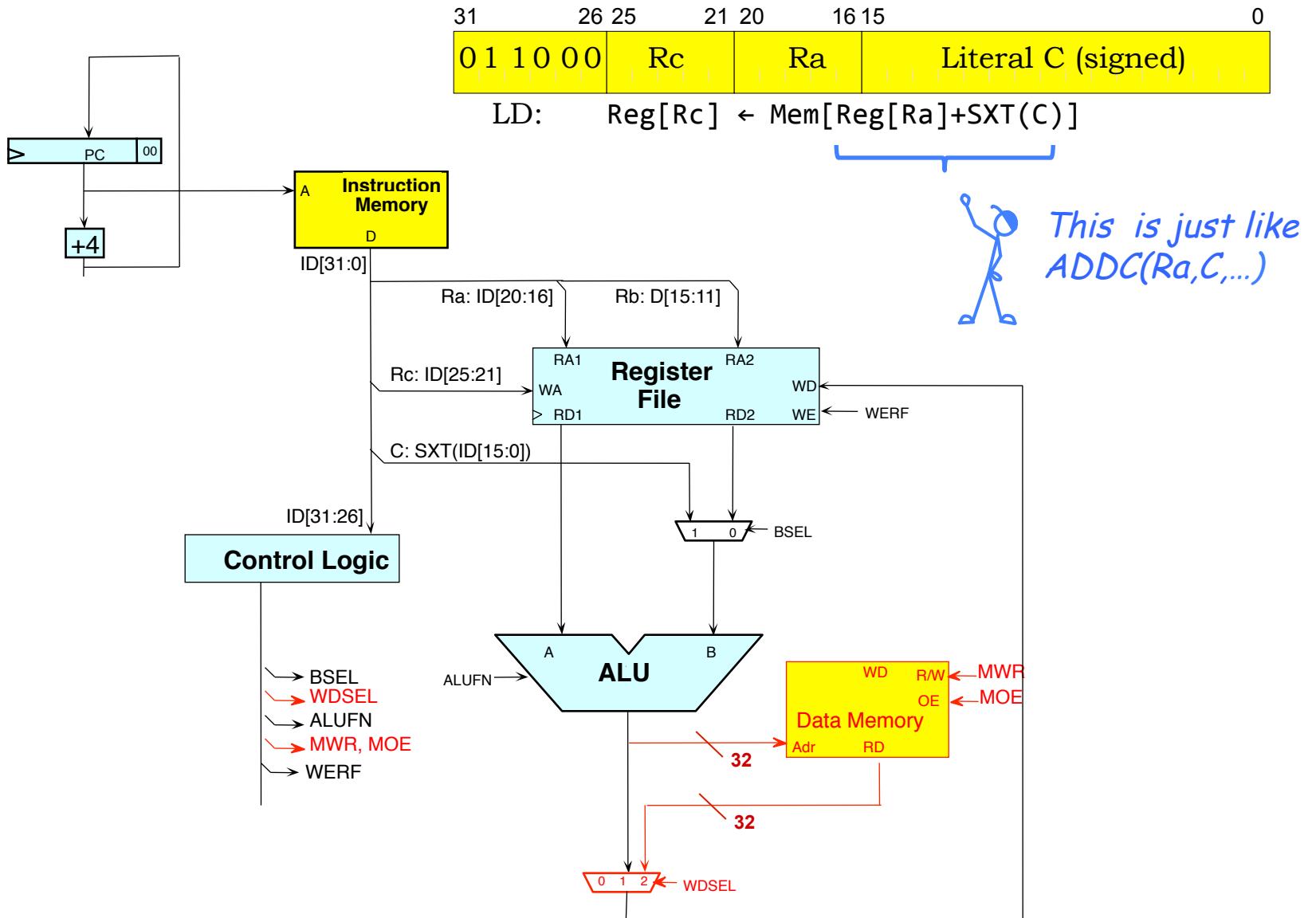
Just replicate ID[15] sixteen times to create high-order bits!



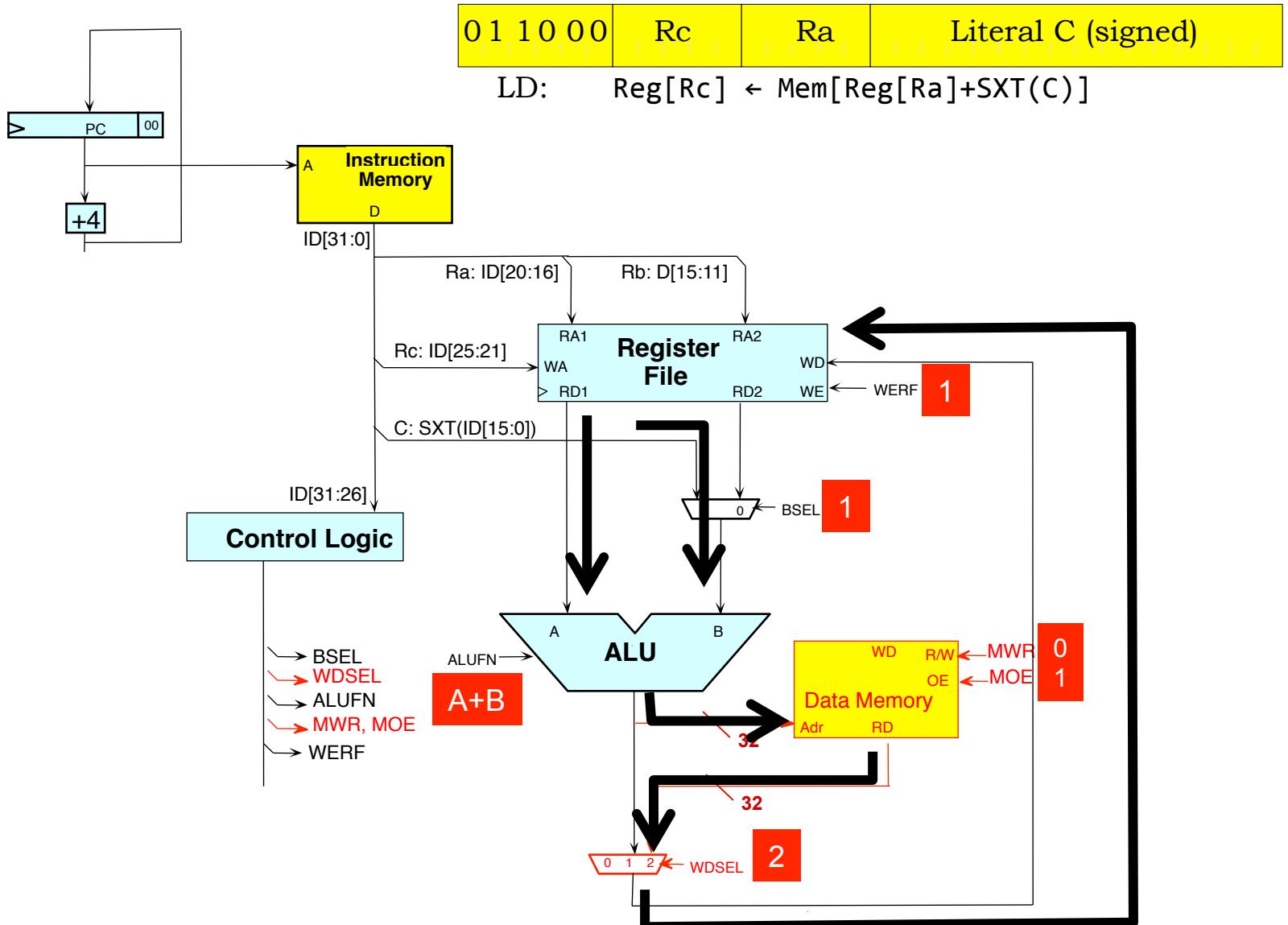
ALU Operations (with constant)



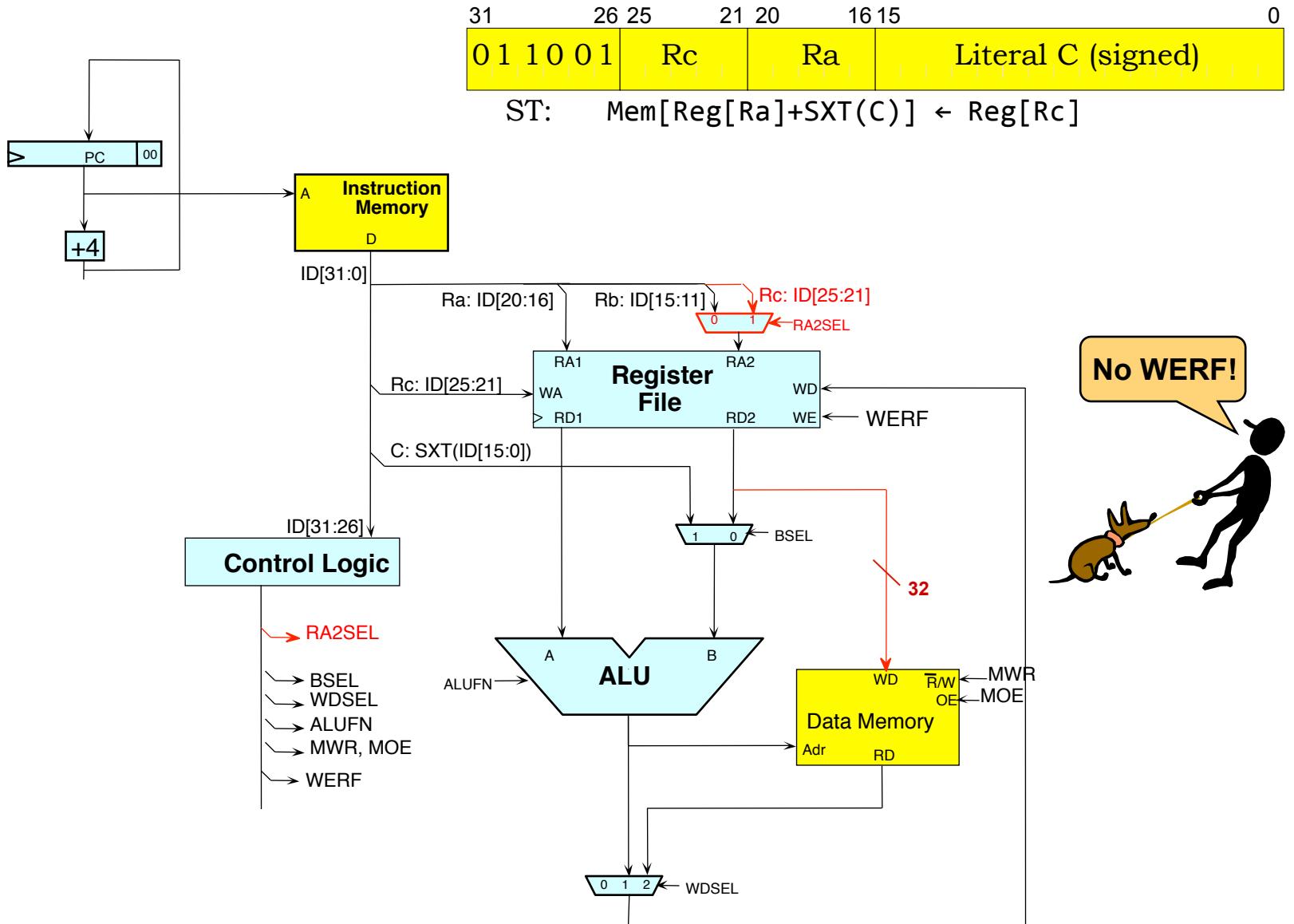
Load Instruction



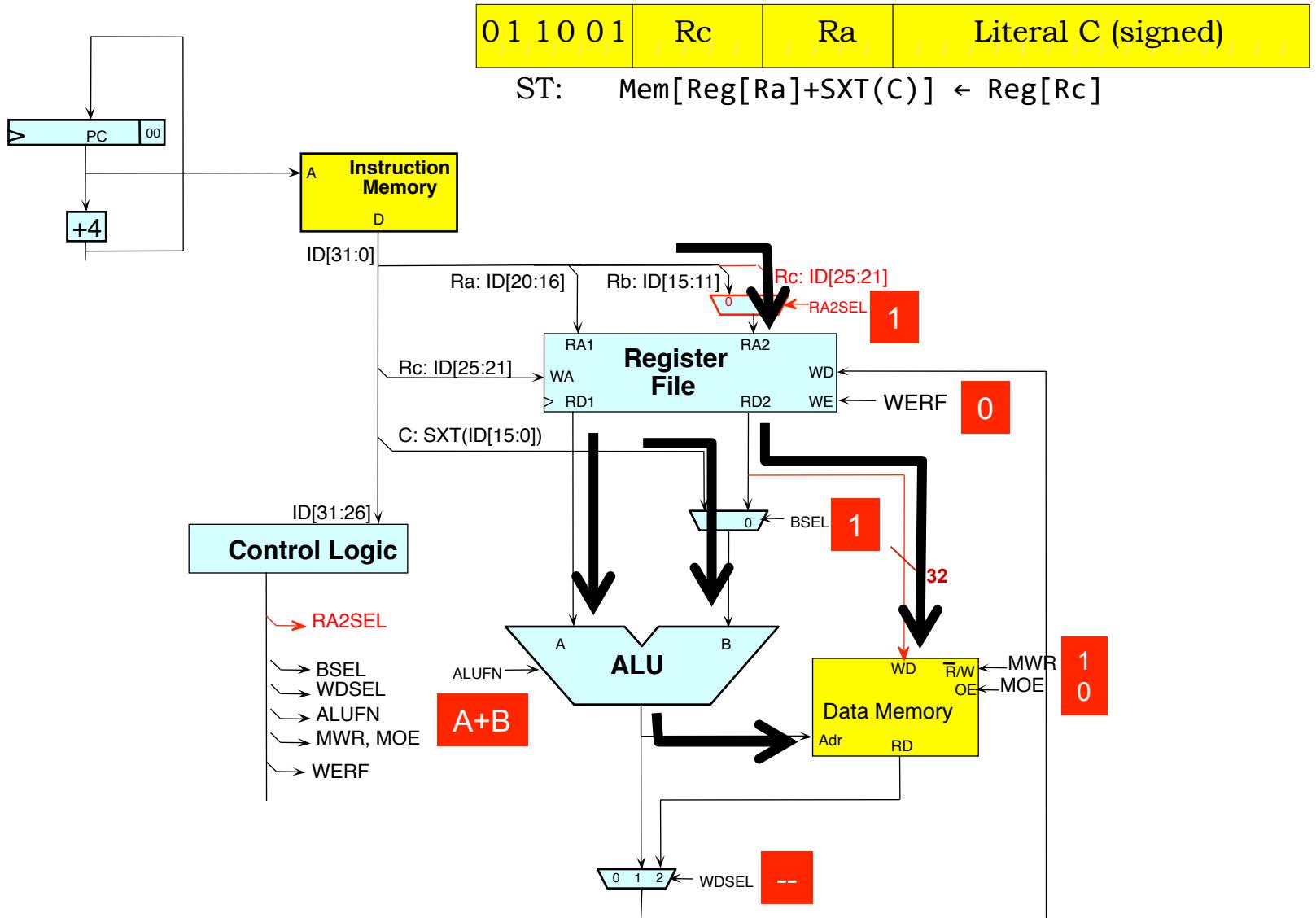
Load Instruction



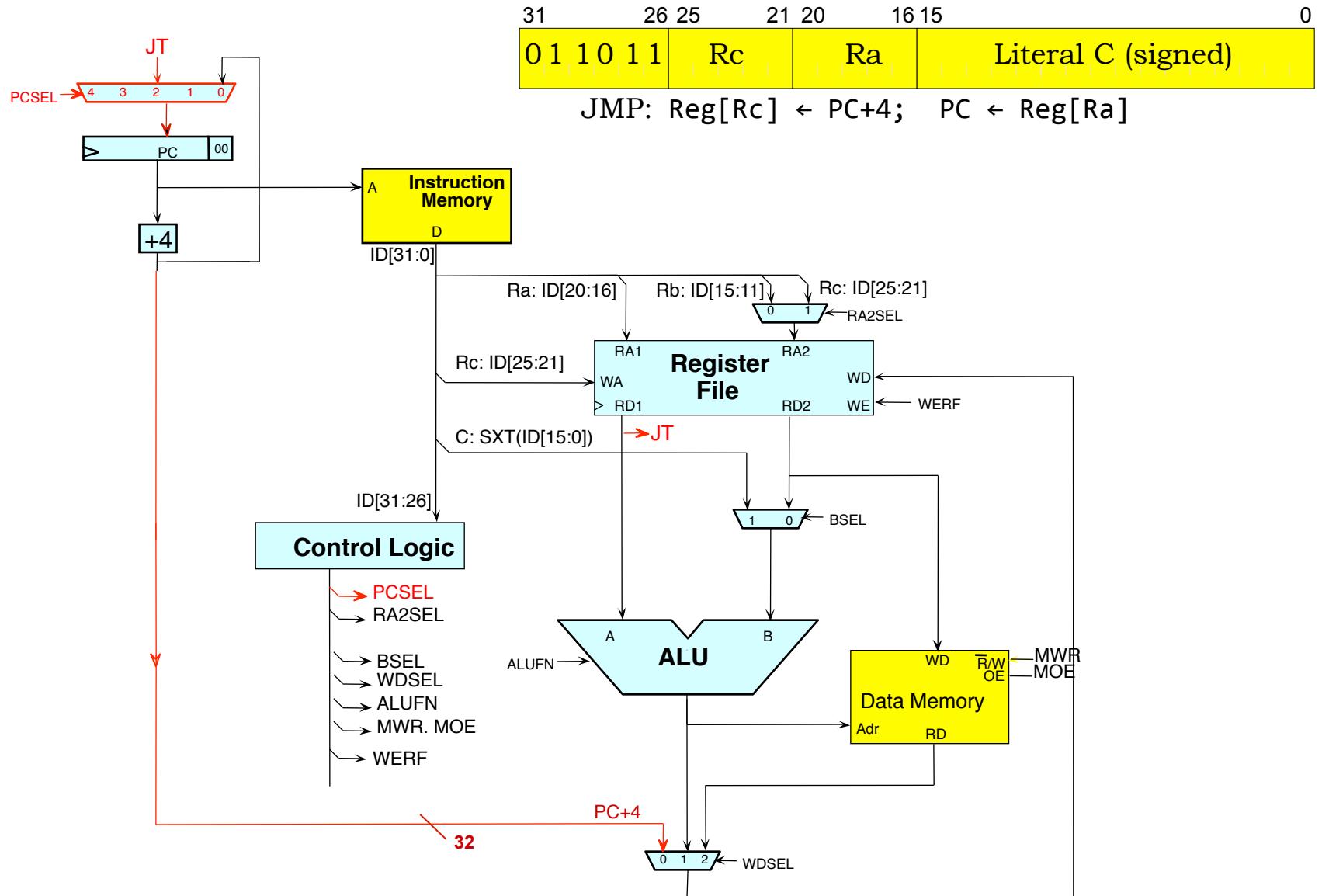
Store Instruction



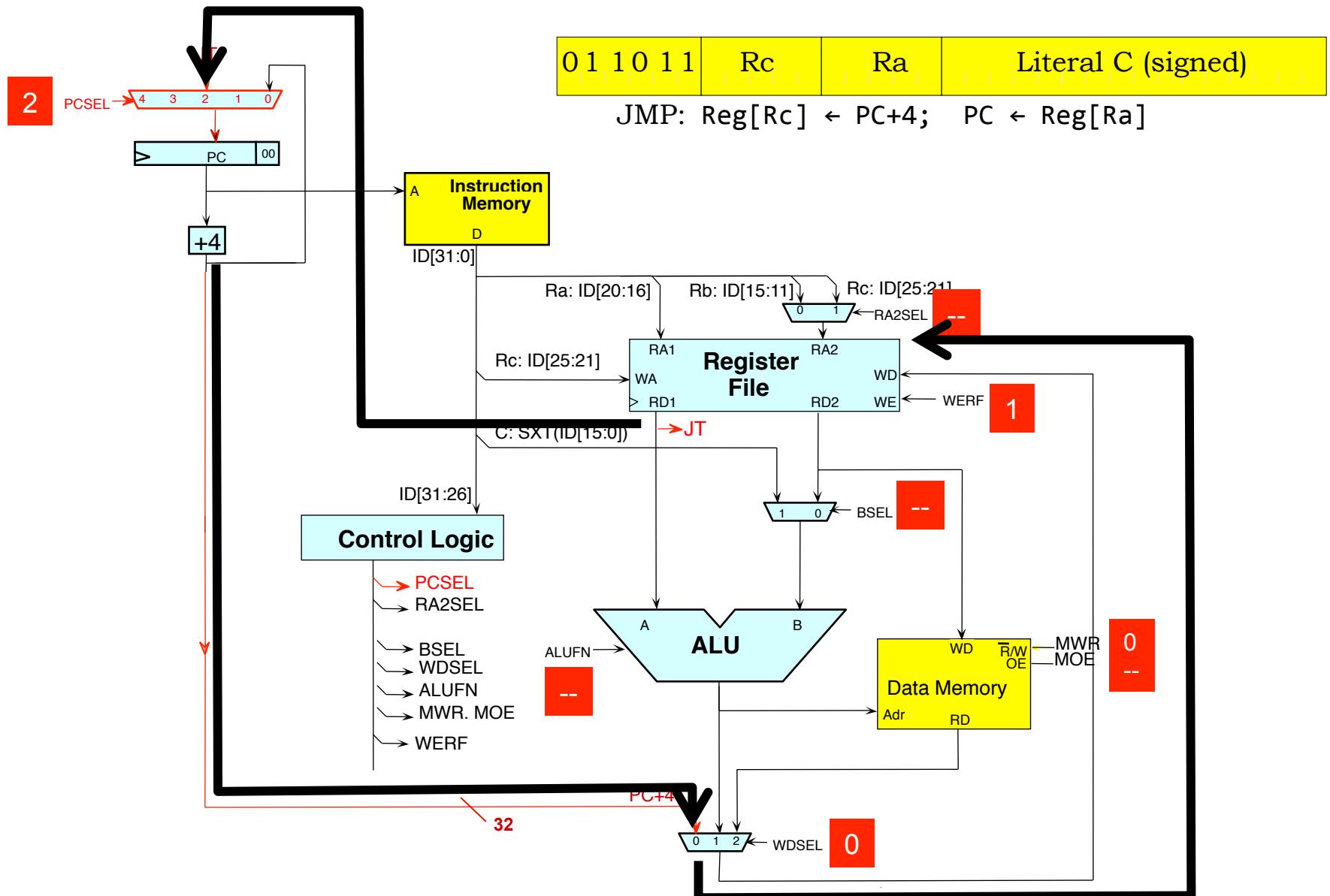
Store Instruction



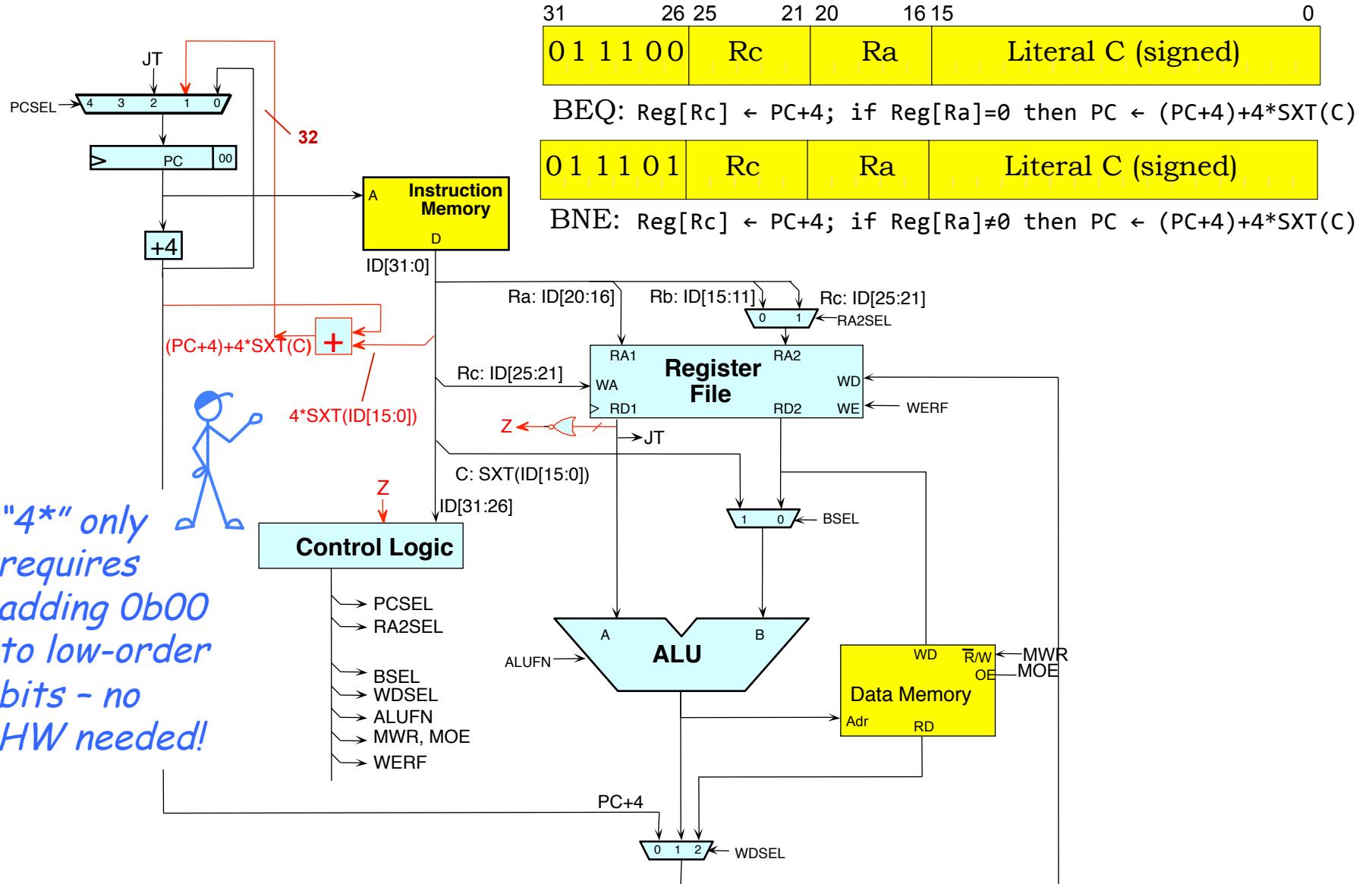
JMP Instruction



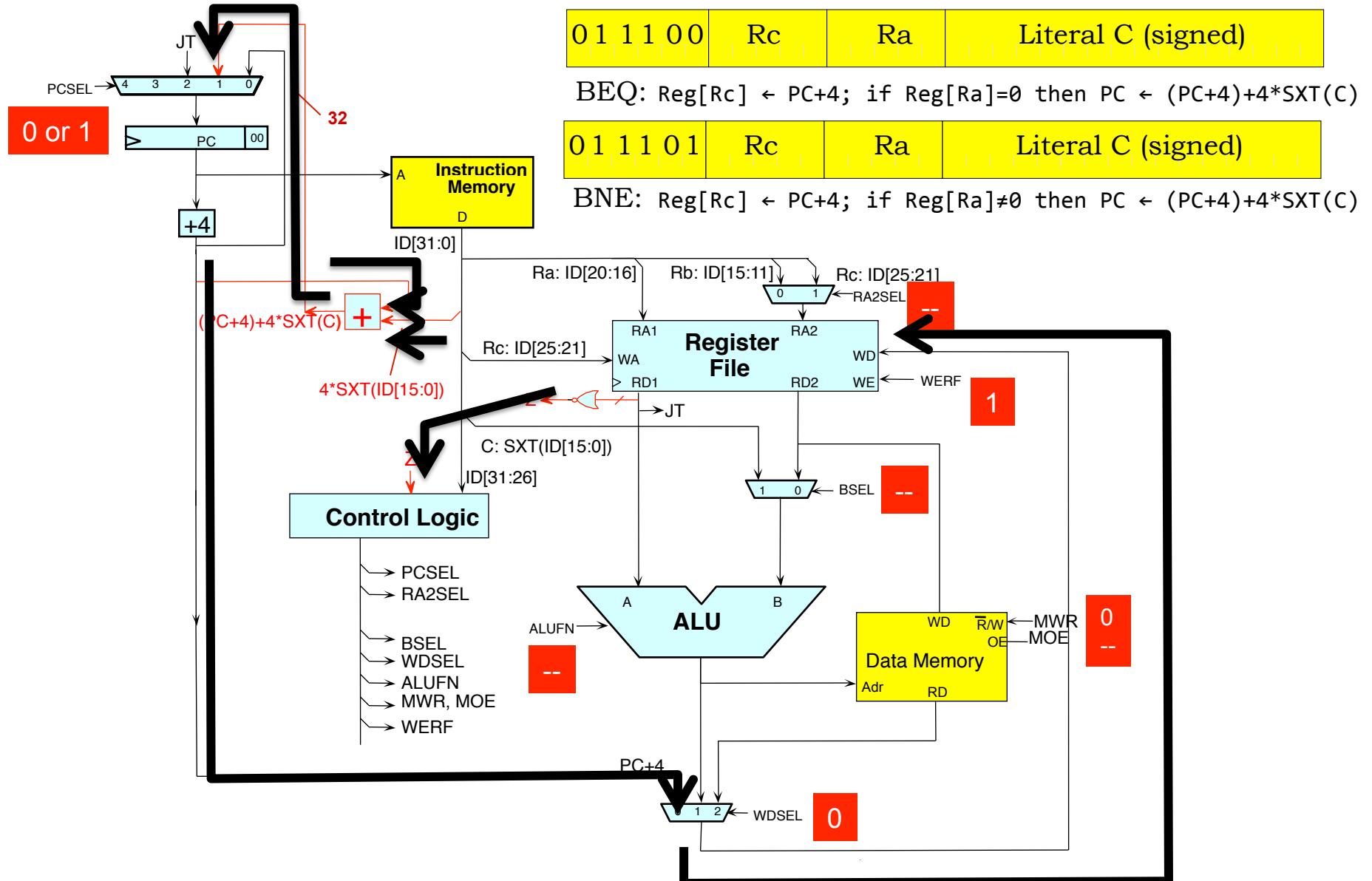
JMP Instruction



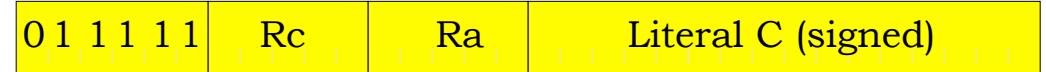
BEQ/BNE Instructions



BEQ/BNE Instructions



Load Relative Instruction



LDR: $\text{Reg}[\text{Rc}] \leftarrow \text{Mem}[\text{PC} + 4 + 4 * \text{SXT}(\text{C})]$

What's Load Relative good for anyway??? I thought

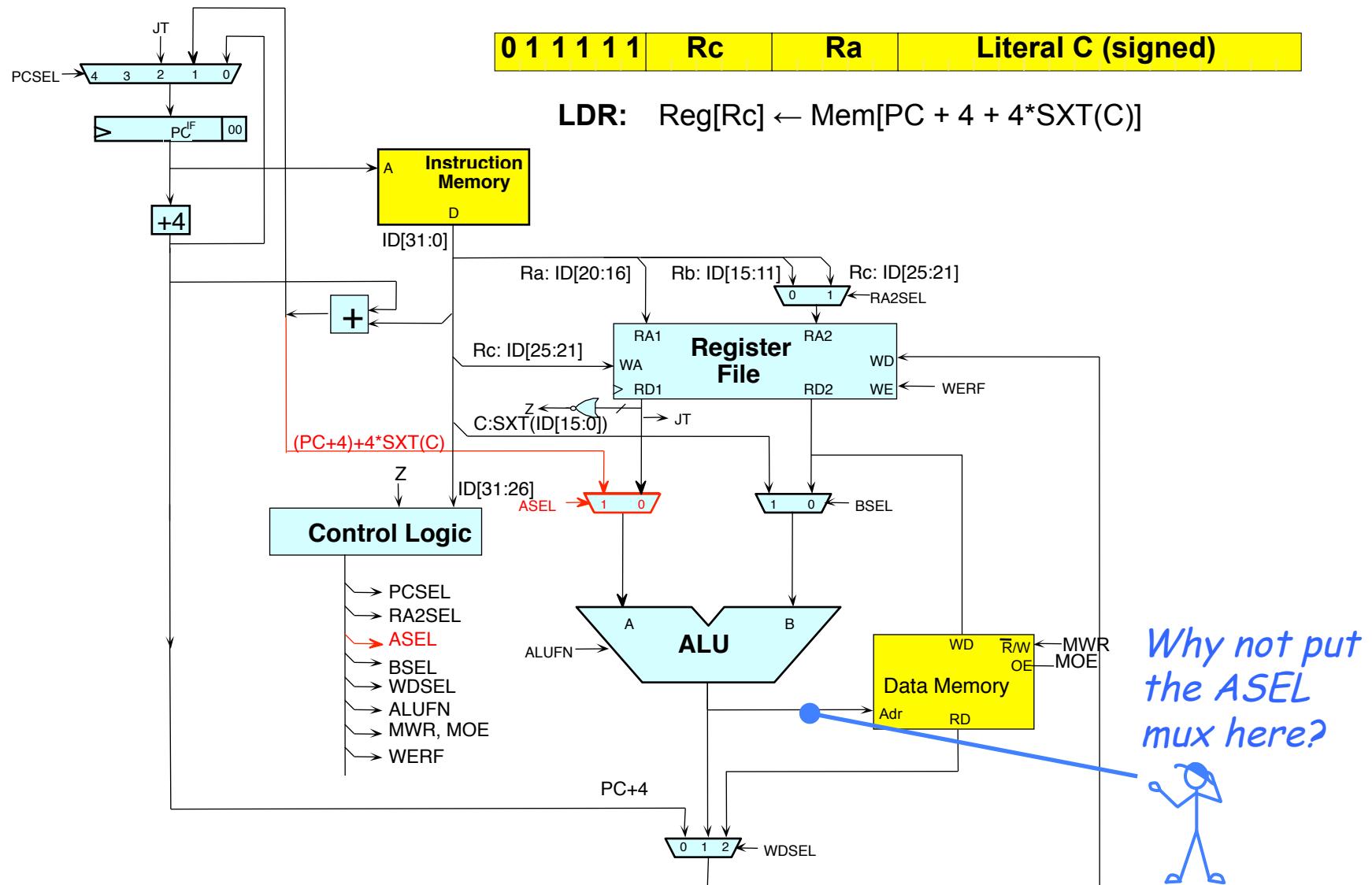
- Code is “PURE”, i.e. READ-ONLY; and stored in a “PROGRAM” region of memory;
- Data is READ-WRITE, and stored either
 - On the STACK (local); or
 - In some GLOBAL VARIABLE region; or
 - In a global storage HEAP.

So why have an instruction designed to load data that's “near” the instruction???

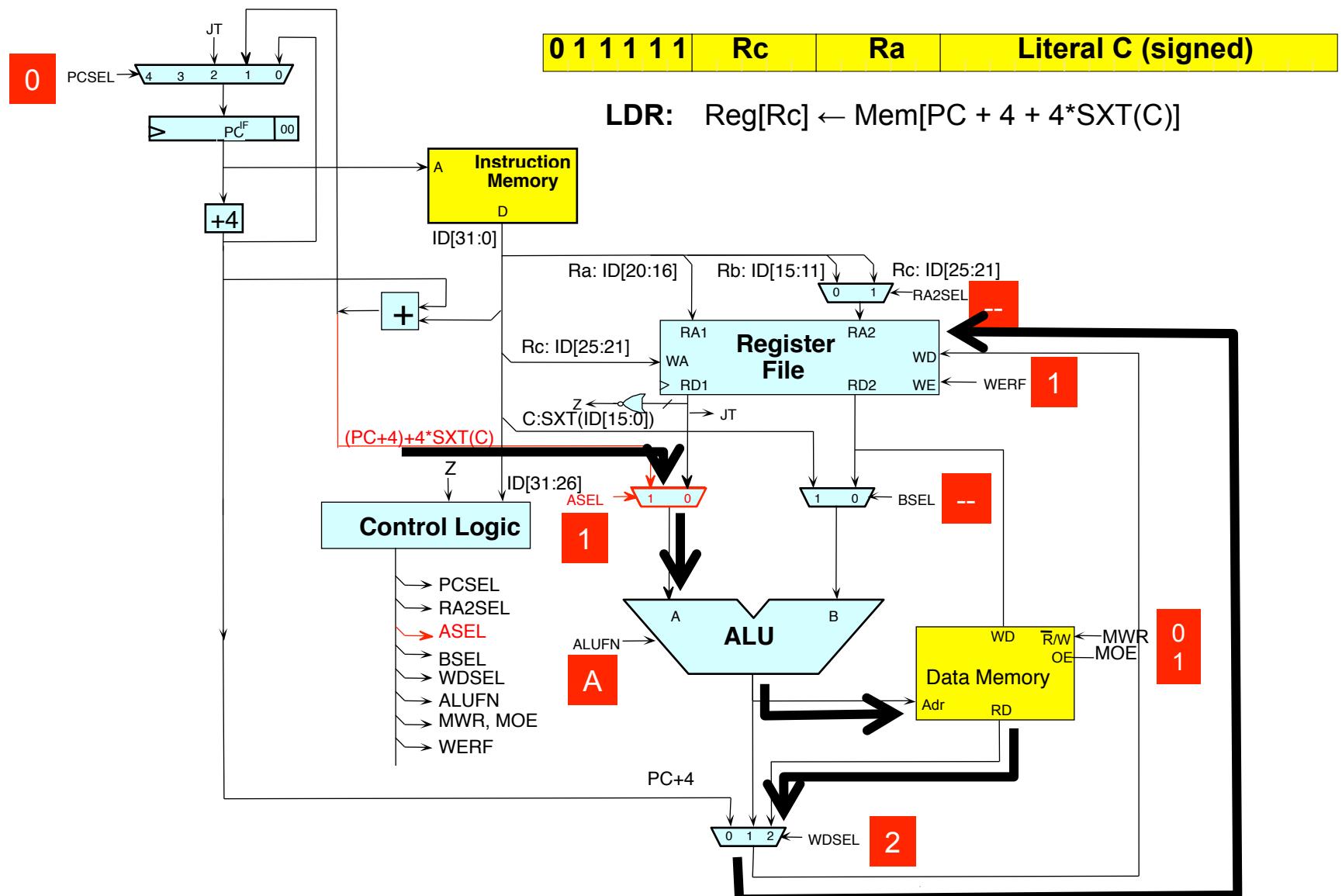
Addresses & other large constants

```
C:     X = X * 123456;  
  
BETA:  
      LD(X, r0)  
      LDR(c1, r1)  
      MUL(r0, r1, r0)  
      ST(r0, X)  
      ...  
c1:    LONG(123456)
```

LDR Instruction



LDR Instruction



Exceptions

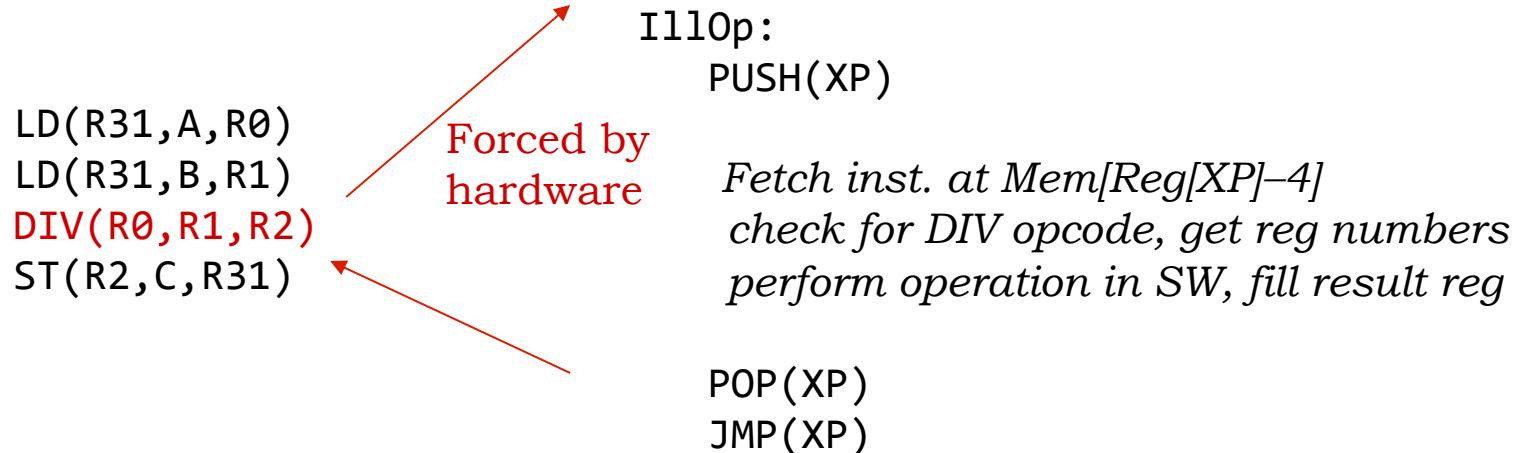
- What if something bad happens?
 - Execution of an illegal opcode
 - Reference to non-existent memory
 - Divide by zero
- Or maybe just something unanticipated
 - User hits a key
 - A packet comes in via the network
- Exceptions let us handle these cases in software:
 - Treat each case as an (implicit) 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

Exception Processing

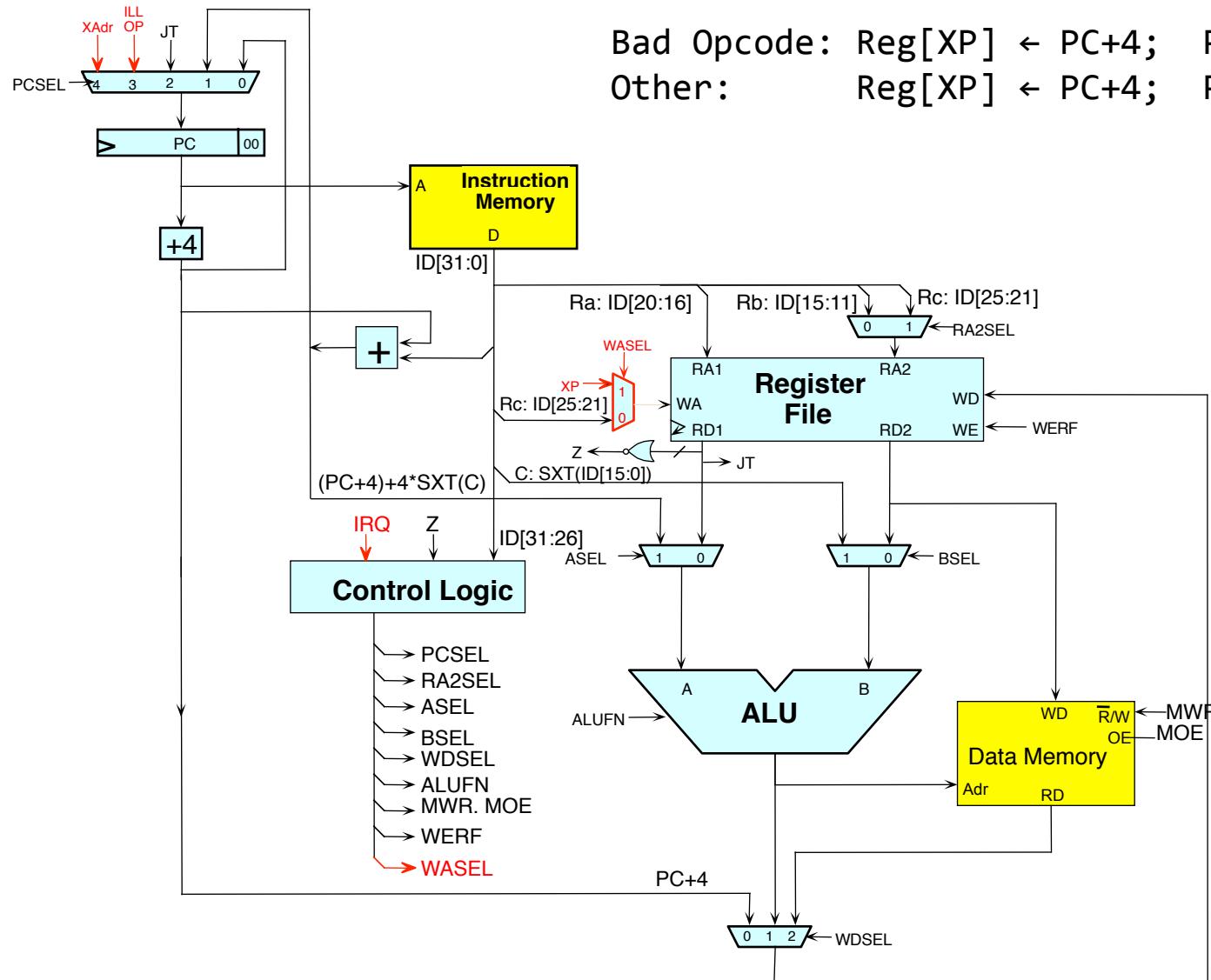
- Plan:
 - Interrupt running program
 - Invoke exception handler (like a procedure call)
 - Return to continue execution
- Exception and interrupt terms often used interchangeably, with minor distinctions:
 - Exceptions usually refer to **synchronous events**, generated by program (e.g., illegal instruction, divide-by-0, illegal address)
 - Interrupts usually refer to **asynchronous events**, generated by I/O devices (e.g., keystroke, packet received, disk transfer complete)

Exception Implementation

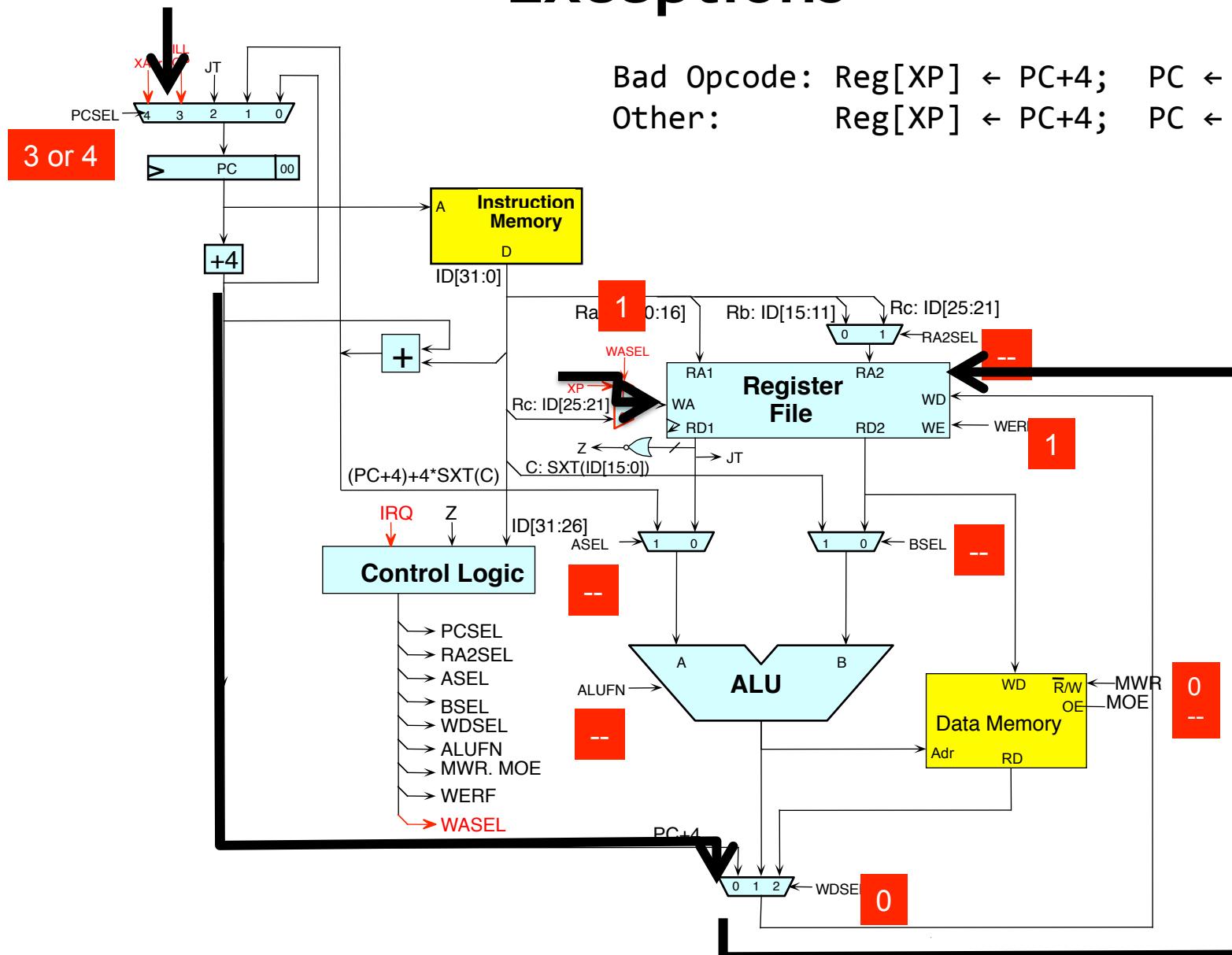
- Instead of executing instruction, fake a procedure call
 - Save current PC+4 (as branches do)
 - Load PC with exception vector: 0x4 for synchronous events, 0x8 for asynchronous events
- We save PC+4 in register R30 (which we call XP)
 - ... and prohibit programs from using XP (why?)
- Example: DIV unimplemented



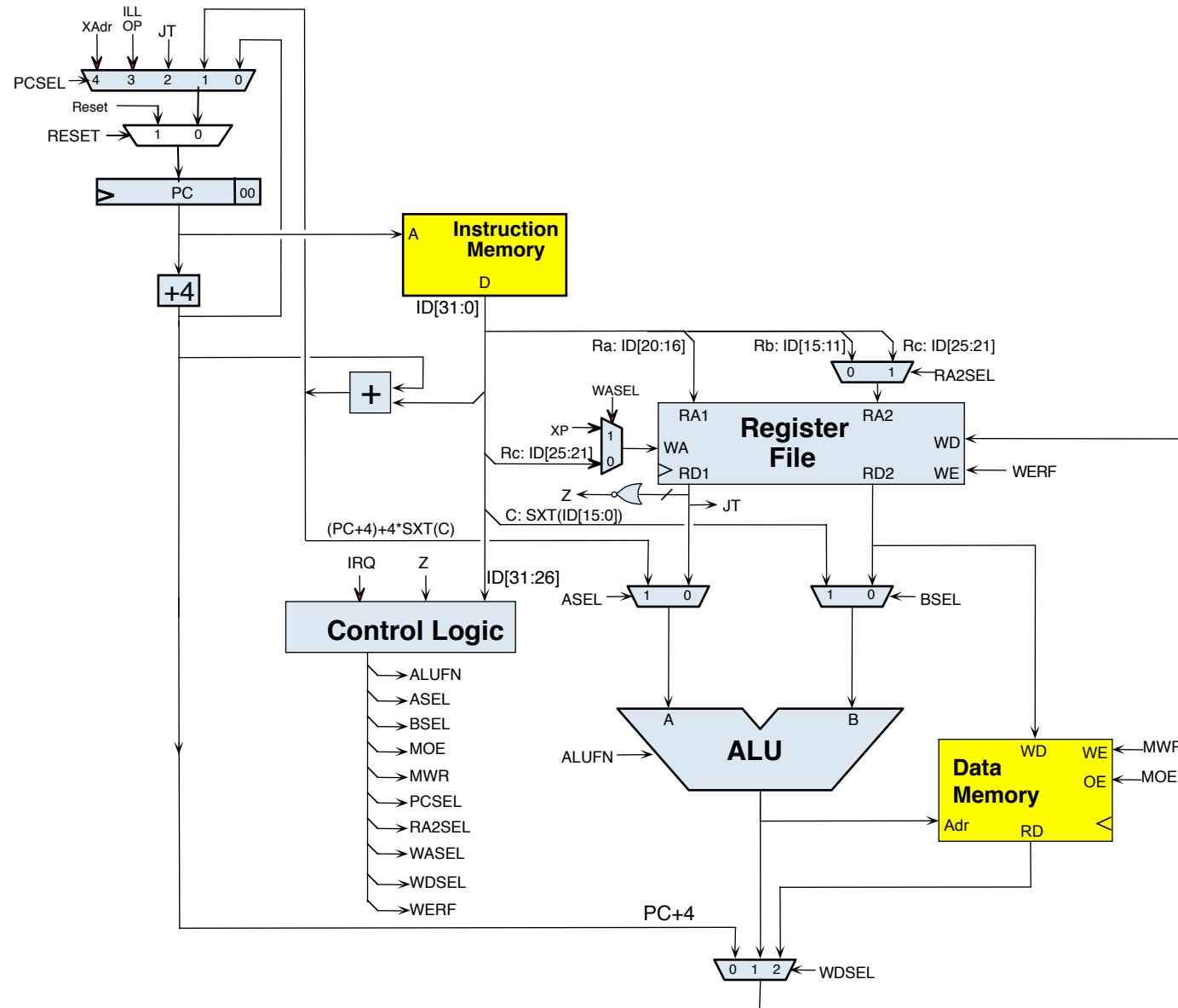
Exceptions



Exceptions



Beta: Our “Final Answer”



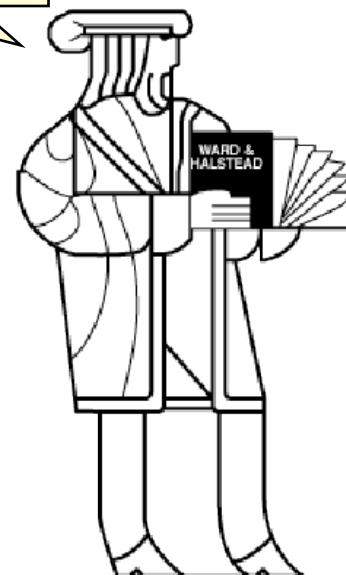
Control Logic

	RESET	IRQ	OP	OPC	LD	LDR	ST	JMP	BEQ	BNF	LLOP
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

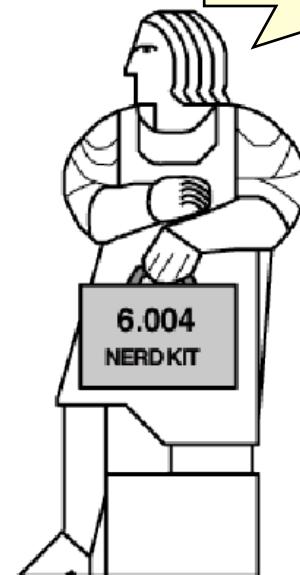
Implementation choices:

- 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!)

*Is **that** all
there is to
building a
processor???*



*No.
You've gotta print
up all those little
“Beta Inside”
stickers.*

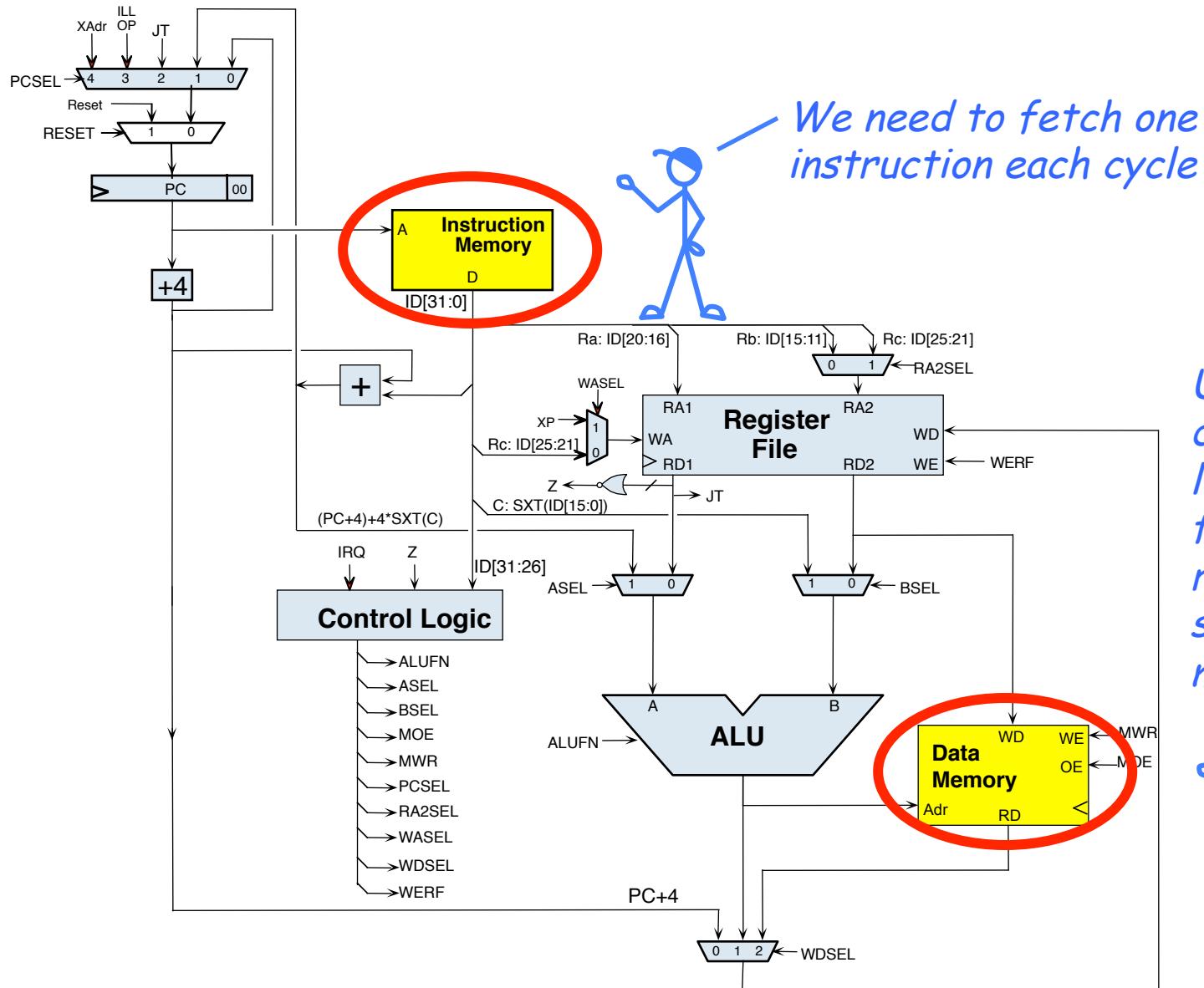


14. Caches & The Memory Hierarchy

6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2016 MIT EECS

Our “Computing Machine”



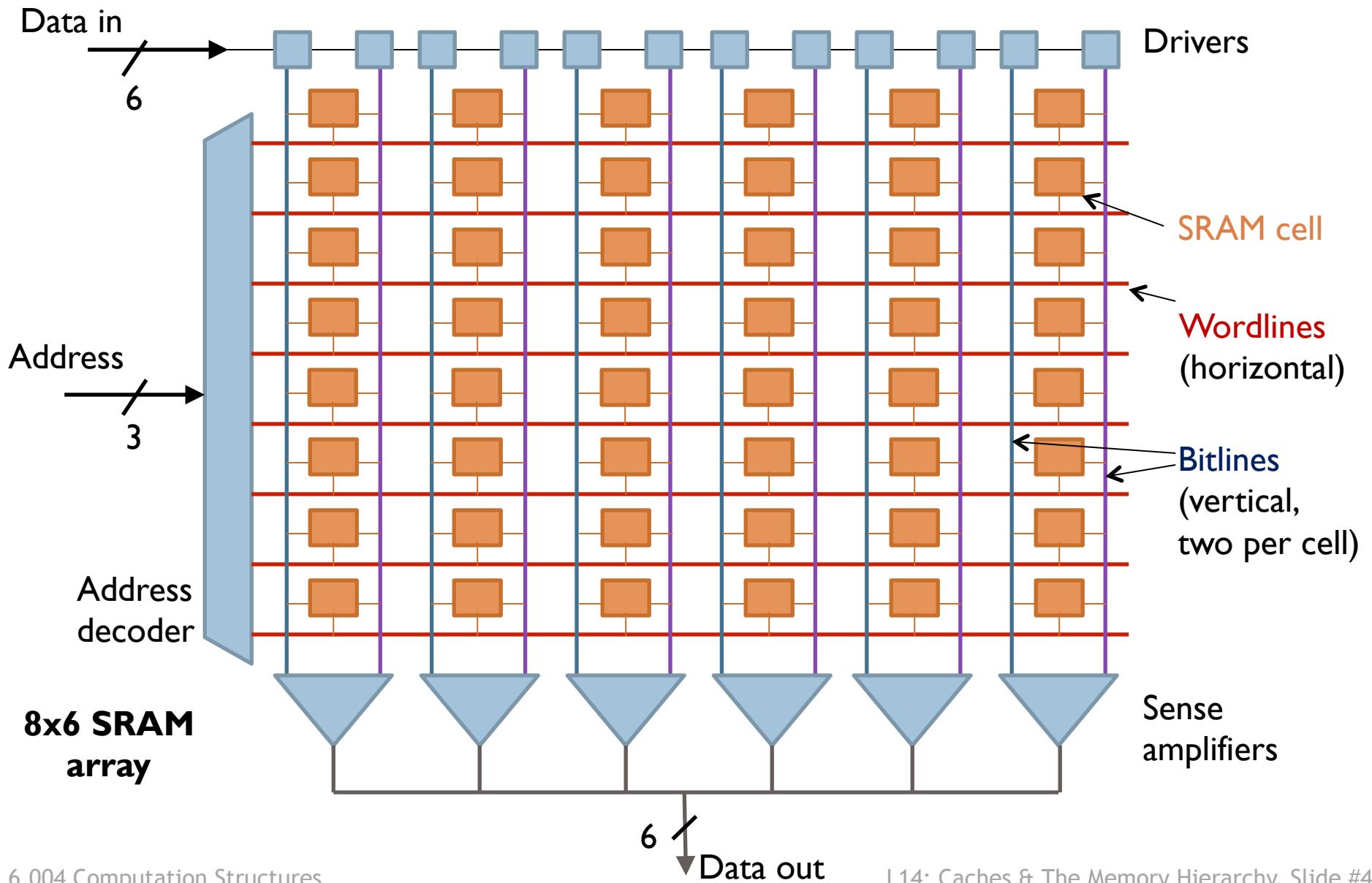
Memory Technologies

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	\$\$\$\$	Processor Datapath
SRAM	~10 KB-10 MB	1-10 ns	~\$1000	Memory Hierarchy
DRAM	~10 GB	80 ns	~\$10	
Flash*	~100 GB	100 us	~\$1	I/O subsystem
Hard disk*	~1 TB	10 ms	~\$0.10	

* non-volatile (retains contents when powered off)

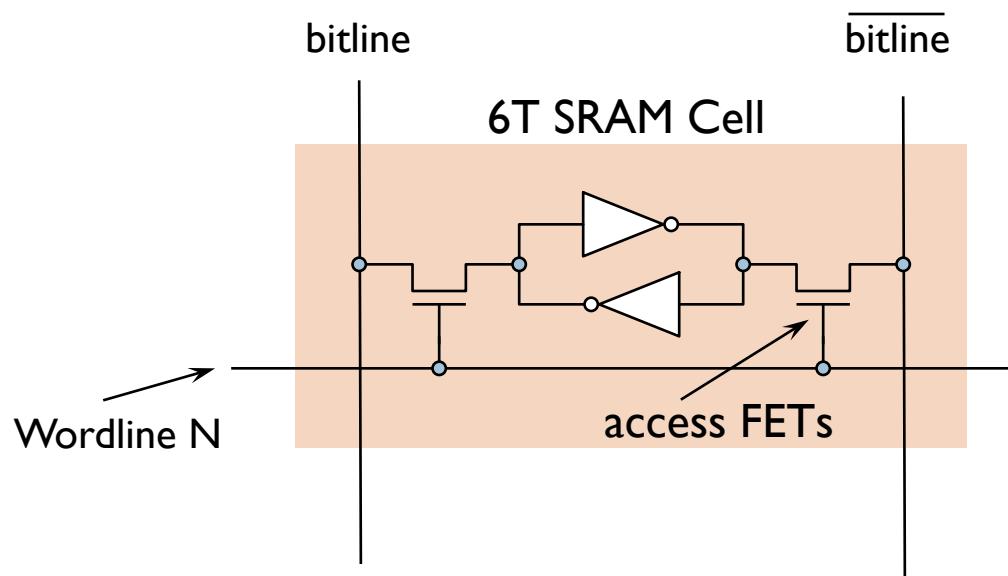
Static RAM (SRAM)



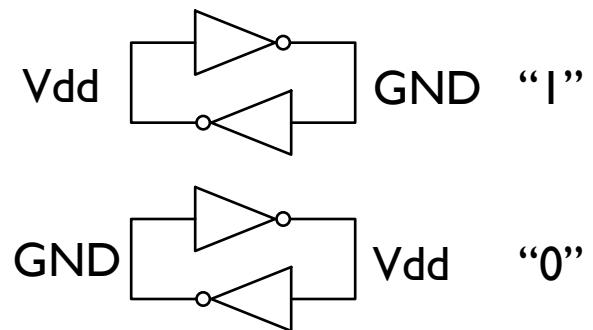
SRAM Cell

6-MOSFET (6T) cell:

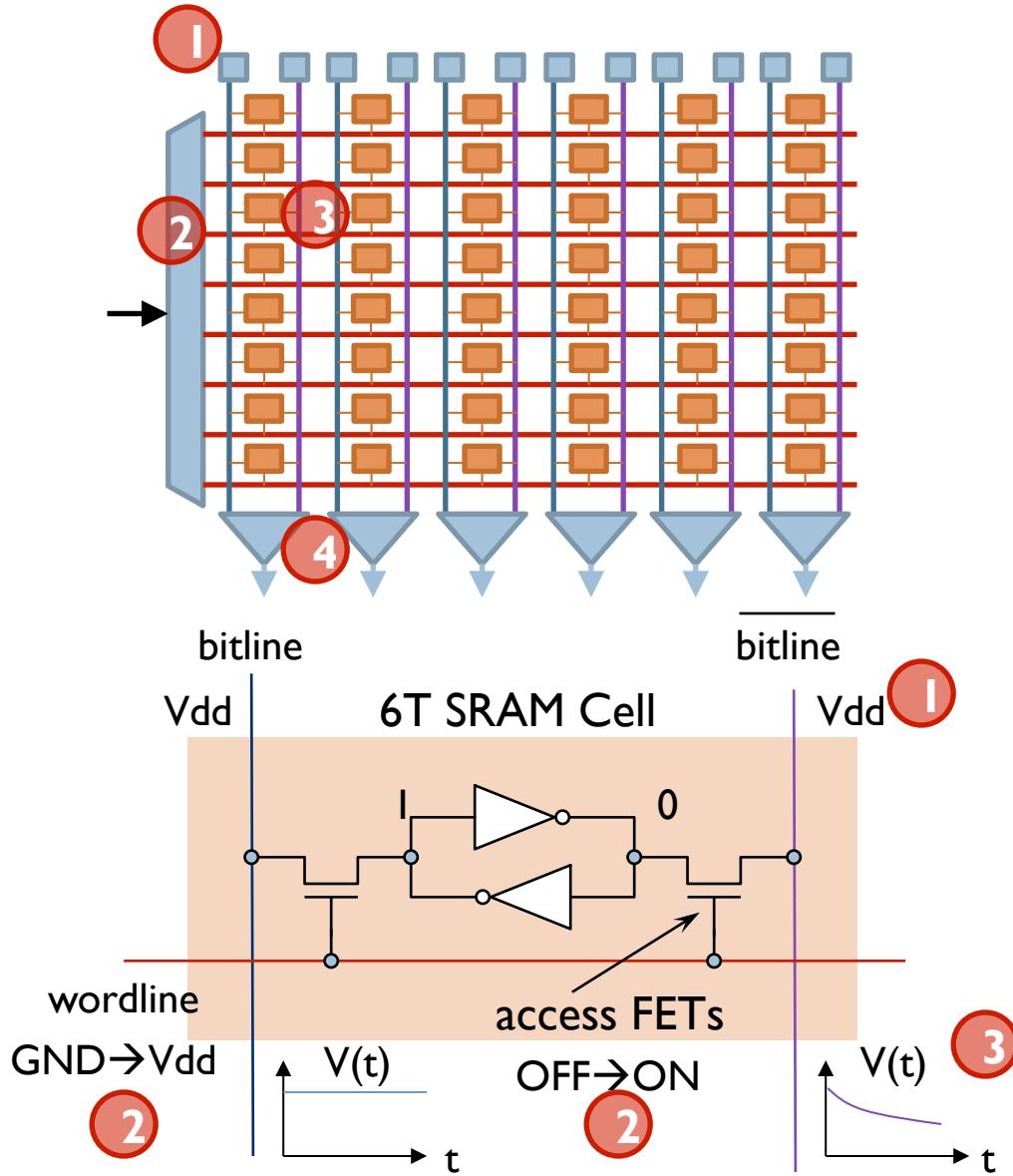
- Two CMOS inverters (4 MOSFETs) forming a **bistable element**
- Two **access transistors**



Bistable element
(two stable states)
stores a single bit

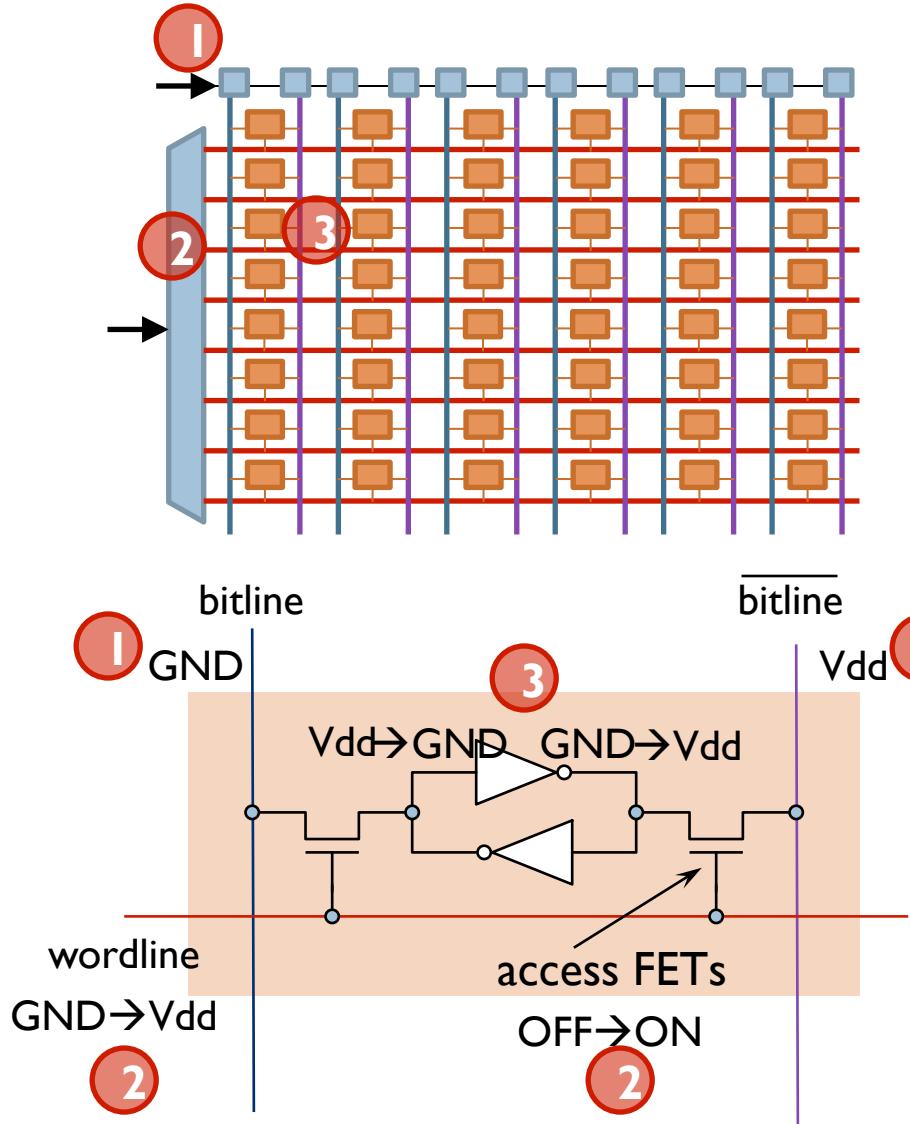


SRAM Read



1. Drivers precharge all bitlines to V_{dd} (1), and leave them floating
2. Address decoder activates one wordline
3. Each cell in the activated word slowly pulls down one of the bitlines to GND (0)
4. Sense amplifiers sense change in bitline voltages, producing output data

SRAM Write

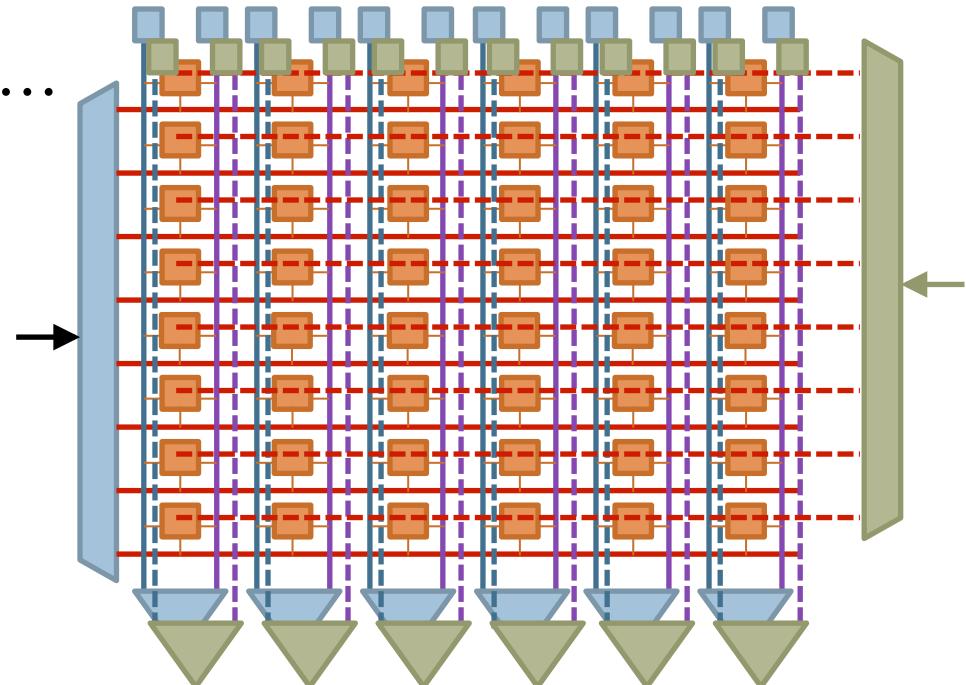


1. Drivers set and hold bitlines to desired values (Vdd and GND for 1, GND and Vdd for 0)
2. Address decoder activates one wordline
3. Each cell in word is overpowered by the drivers, stores value

All transistors are carefully sized so that bitline GND overpowers cell Vdd, but bitline Vdd does not overpower cell GND (why?)

Multiported SRAMs

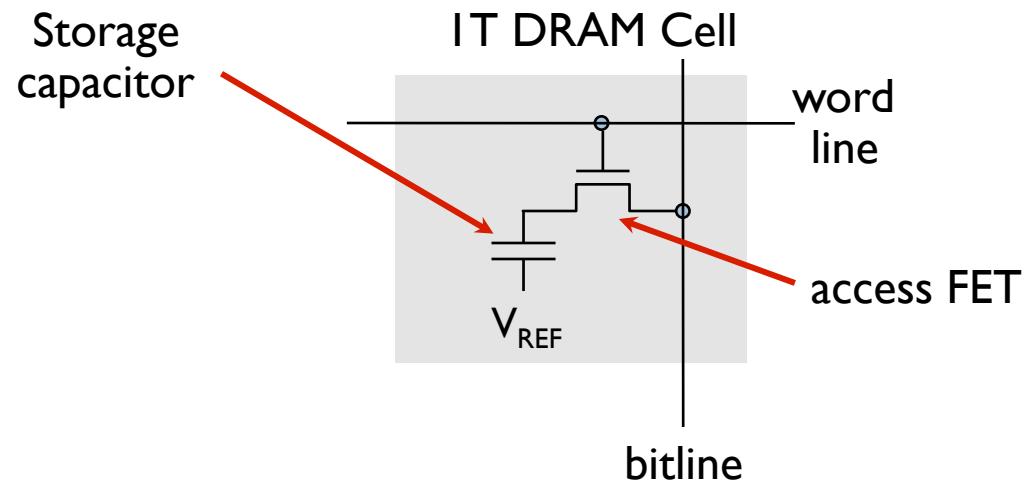
- SRAM so far can do either one read or one write/cycle
- We can do multiple reads and writes with multiple ports by adding one set of wordlines and bitlines per port
- Cost/bit? For N ports...
 - Wordlines: $\frac{N}{2^*N}$
 - Bitlines: $\frac{2^*N}{2^*N}$
 - Access FETs: $\frac{2^*N}{2^*N}$
- Wires often dominate area $\rightarrow O(N^2)$ area!



Summary: SRAMs

- Array of $k*b$ 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?

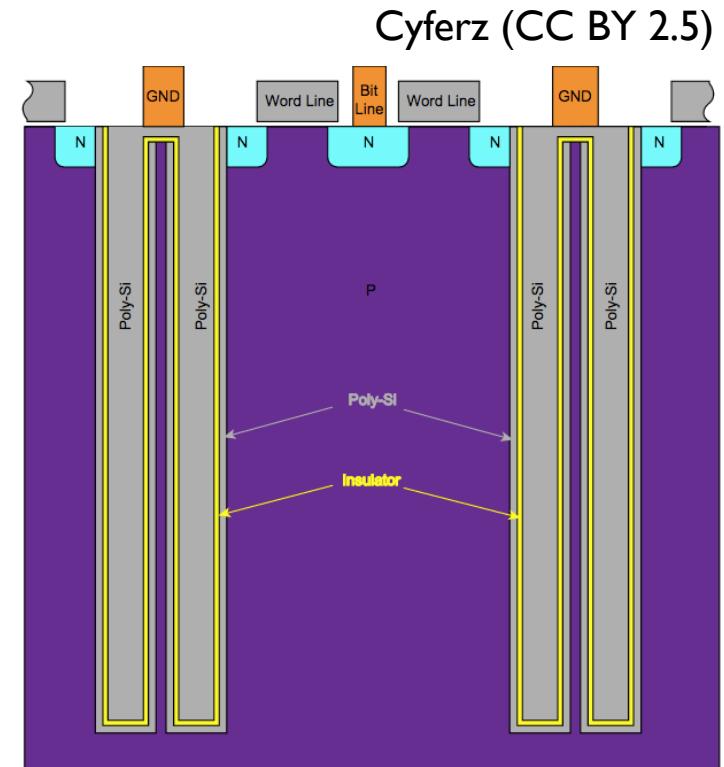
1T Dynamic RAM (DRAM) Cell



C in storage capacitor determined by:

$$C = \frac{e A}{d}$$

better dielectric more area
 thinner film



Trench capacitors
take little area

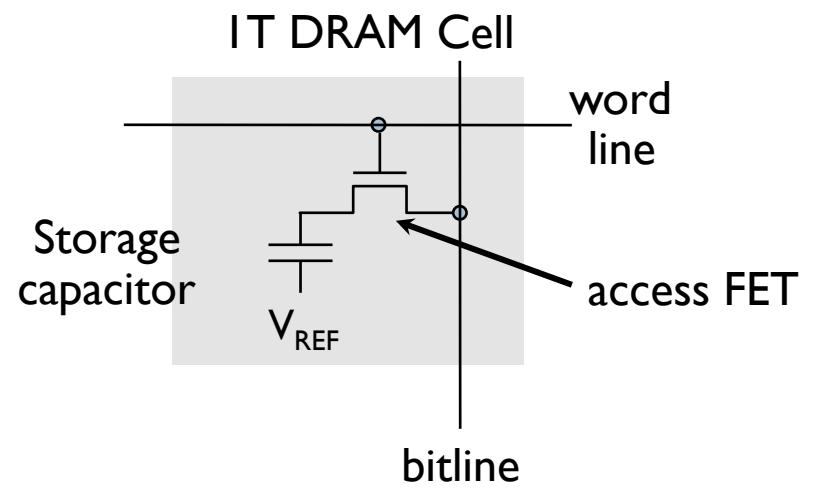
✓ ~20x smaller area than SRAM cell → Denser and cheaper!

✗ Problem: Capacitor leaks charge, must be refreshed periodically (~milliseconds)

DRAM Writes and Reads

- Writes: Drive bitline to Vdd or GND, activate wordline, charge or discharge capacitor

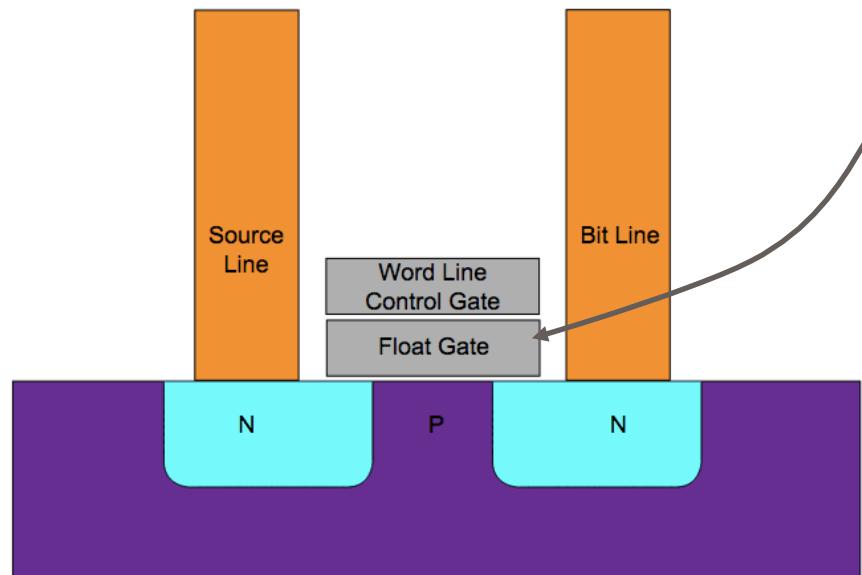
- Reads:
 1. Precharge bitline to $V_{DD}/2$
 2. Activate wordline
 3. Capacitor and bitline share charge
 - If capacitor was discharged, bitline voltage decreases slightly
 - If capacitor was charged, bitline voltage increases slightly
 4. Sense bitline to determine if 0 or 1
 - Issue: **Reads are destructive!** (charge is gone!)
 - So, data must be rewritten to cell at end of read



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

Non-Volatile Storage: Flash



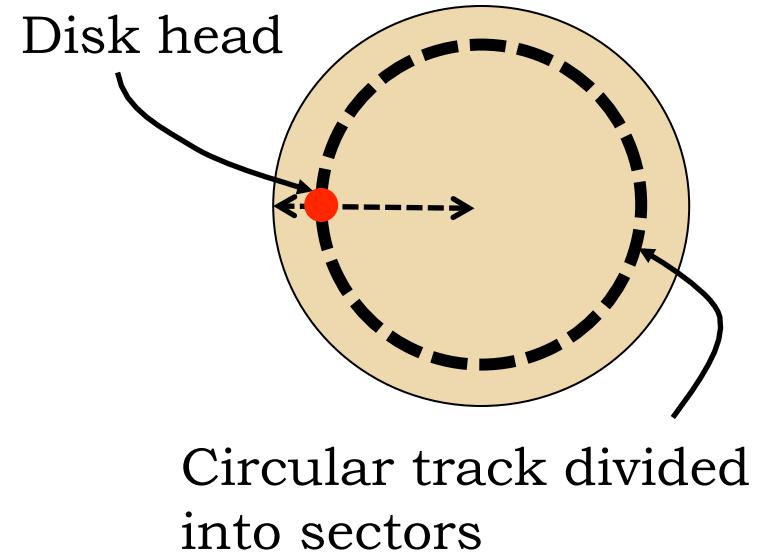
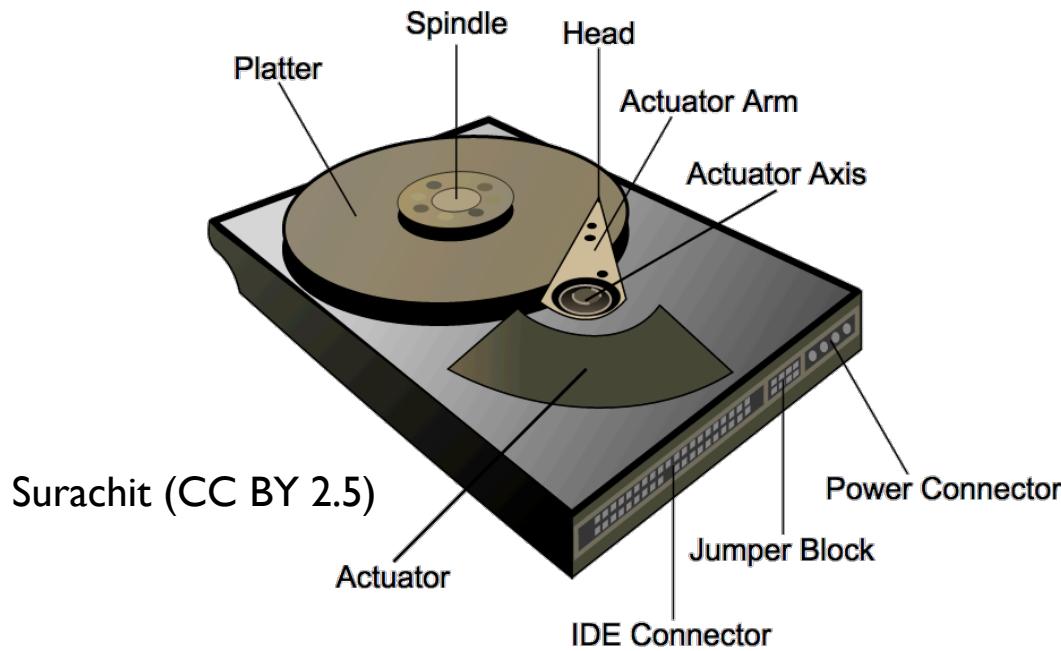
Electrons here diminish strength of field from control gate \Rightarrow no inversion \Rightarrow NFET stays off even when word line is high.

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 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**

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)

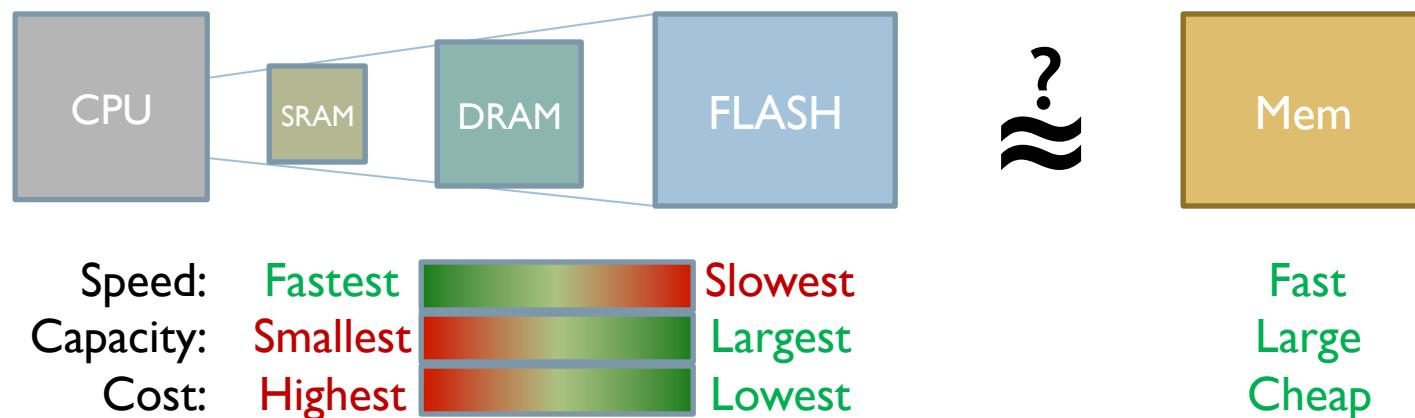
The Memory Hierarchy

Want large, fast, and cheap memory, but...

Large memories are slow (even if built with fast components)

Fast memories are expensive

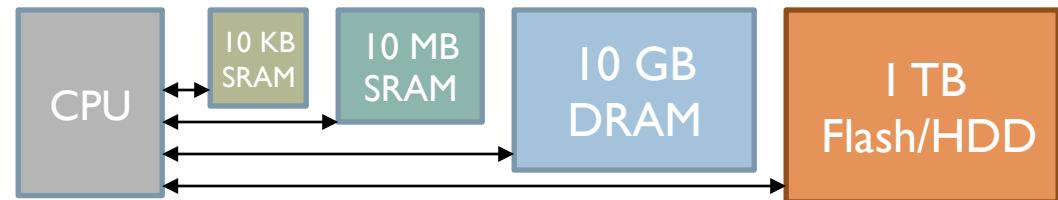
Idea: Can we use a **hierarchical system** of memories with different tradeoffs to **emulate** a large, fast, cheap memory?



Memory Hierarchy Interface

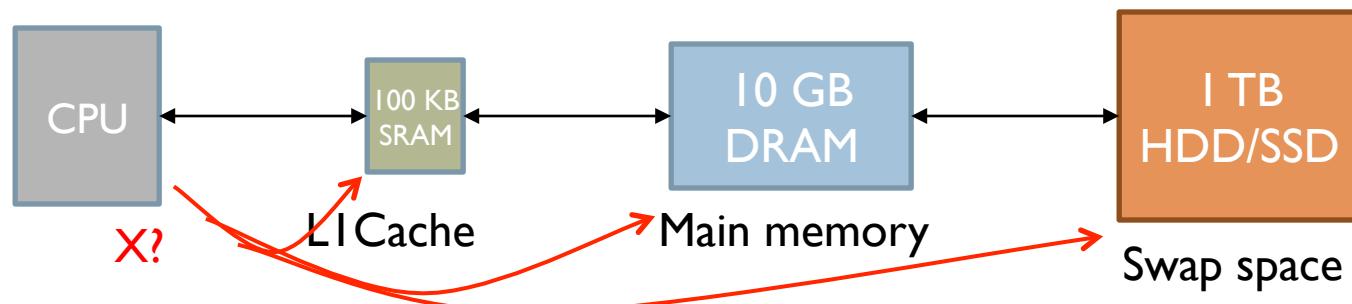
Approach 1: Expose Hierarchy

- 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



The Locality Principle

Keep the most often-used data in a small, fast SRAM (often local to CPU chip)

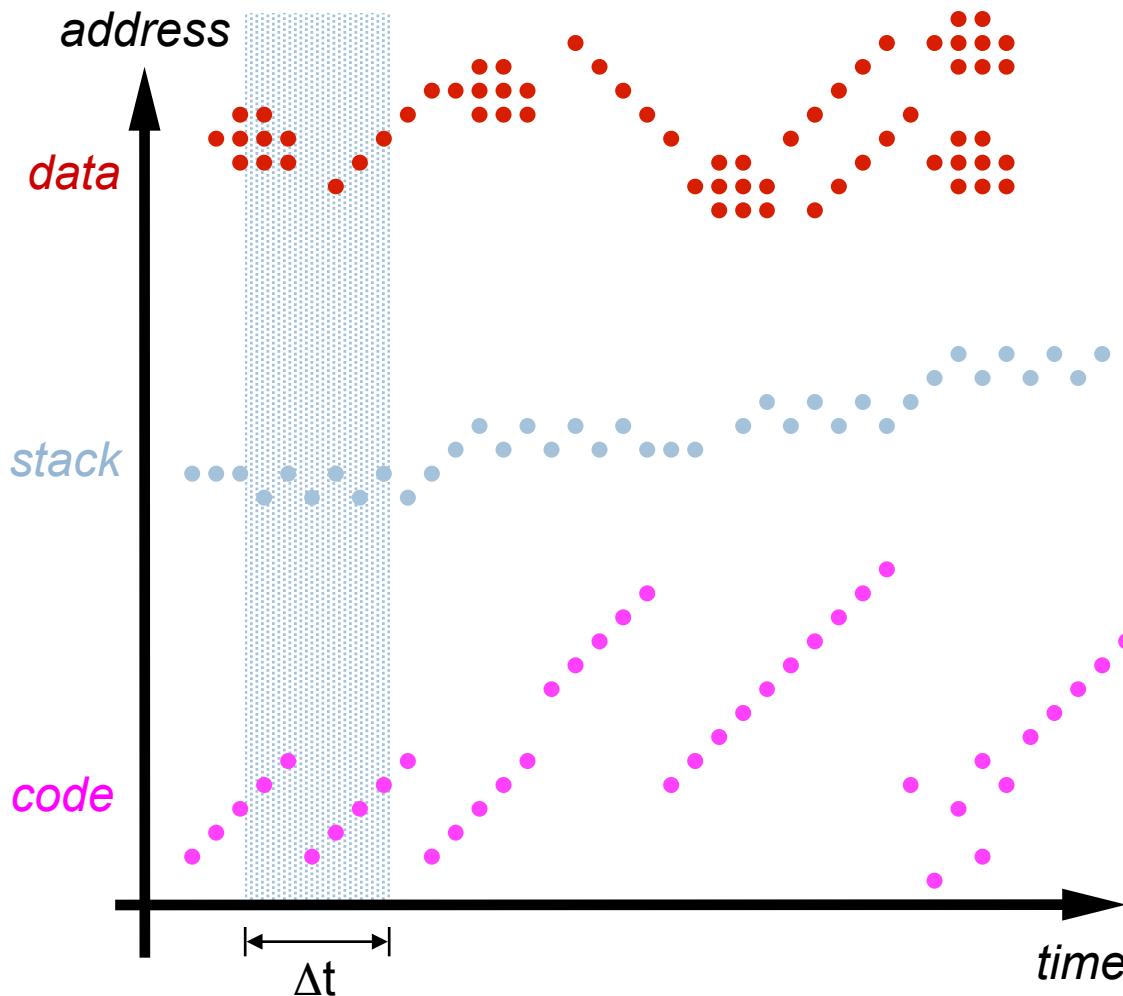
Refer to Main Memory only rarely, for remaining data.

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.

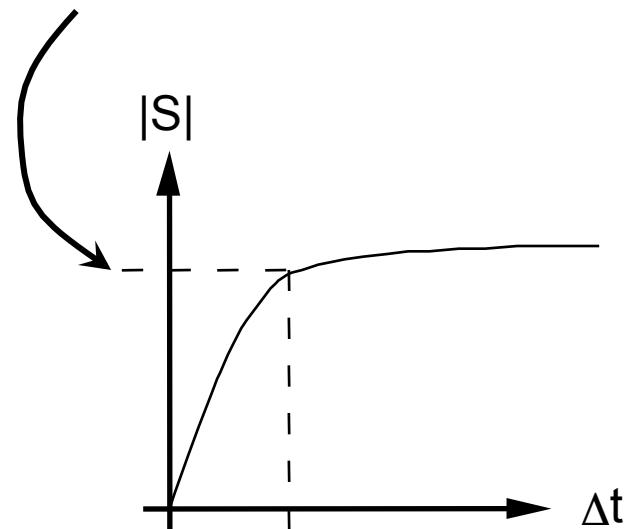
Memory Reference Patterns



S is the set of locations accessed during Δt .

Working set: a set S which changes slowly wrt access time.

Working set size, $|S|$



Caches

Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations

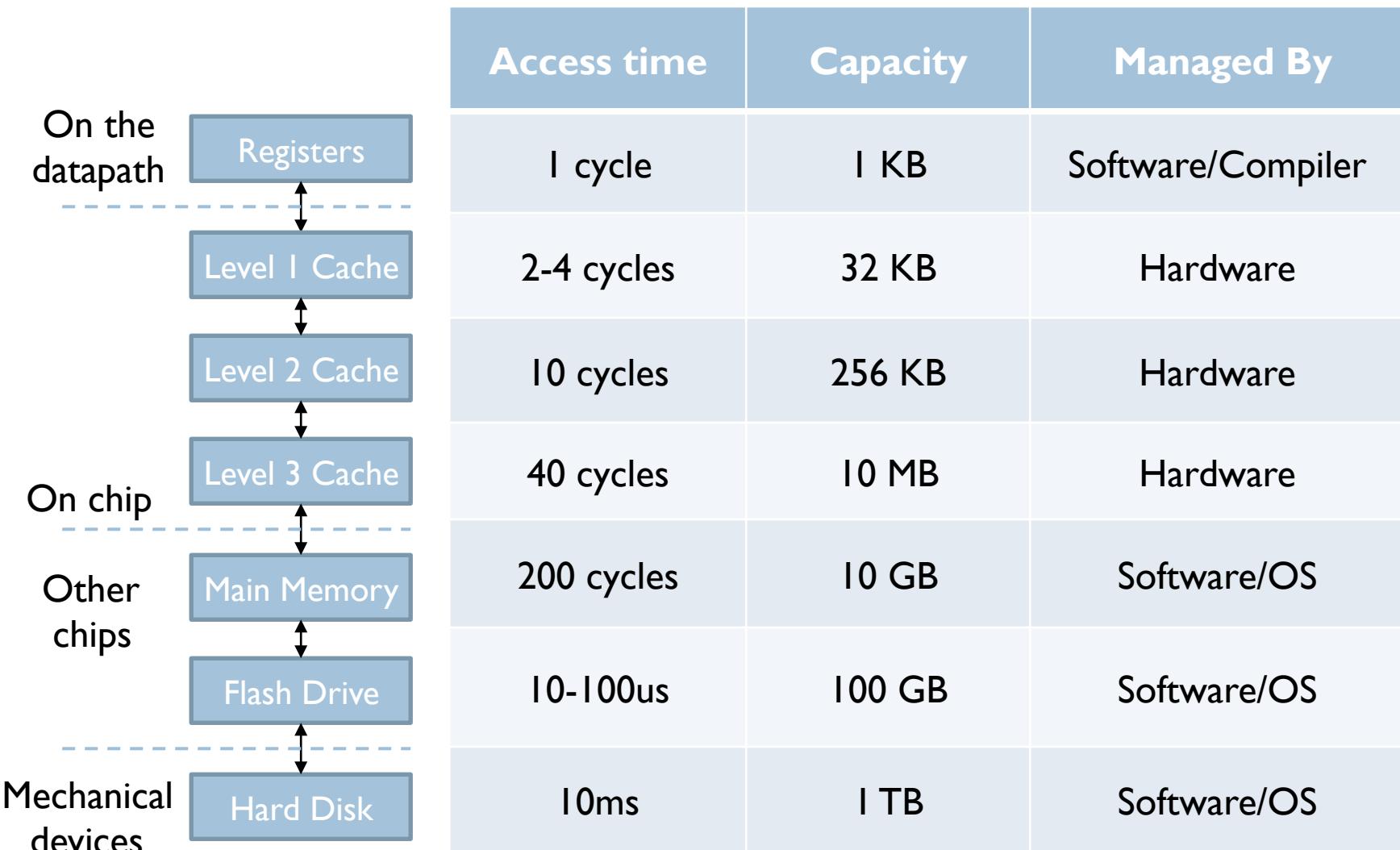
- Very fast access if data is cached, otherwise accesses slower, larger cache or memory
- Exploits the locality principle

Computer systems often use multiple levels of caches

Caching widely applied beyond hardware (e.g., web caches)

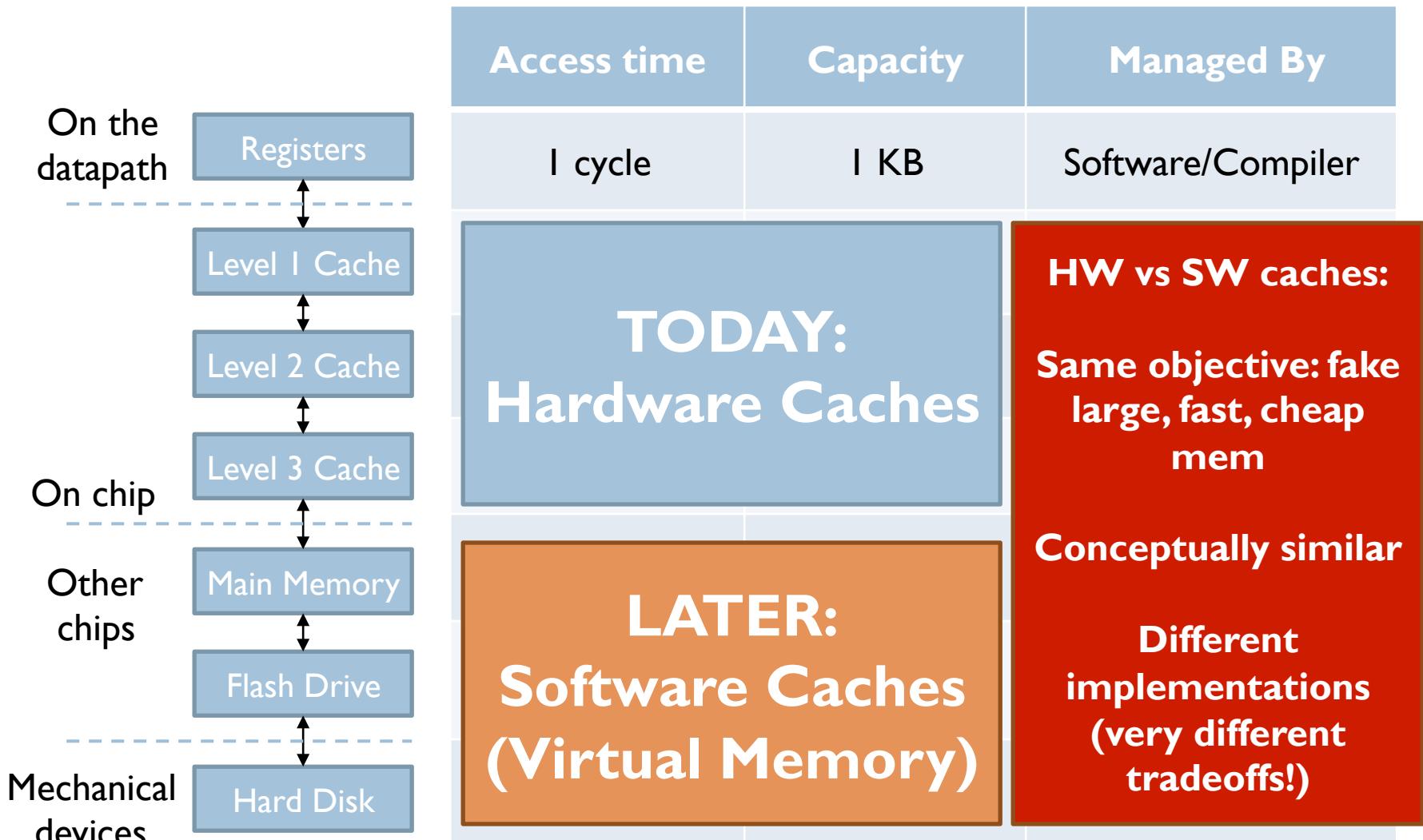
A Typical Memory Hierarchy

- Everything is a cache for something else...

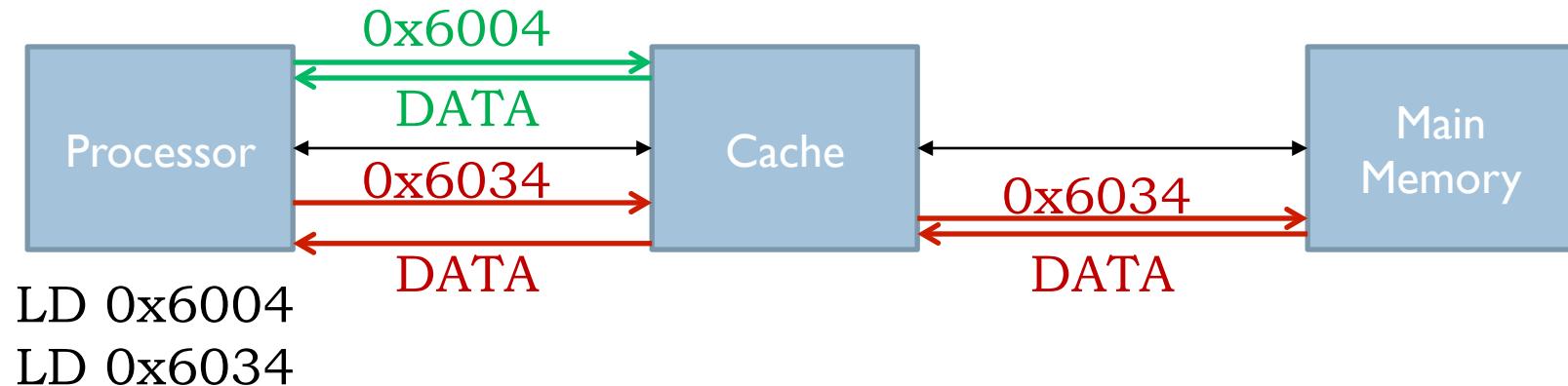


A Typical Memory Hierarchy

- Everything is a cache for something else...



Cache Access



- Processor sends address to cache
- Two options:
 - **Cache hit:** Data for this address in cache, returned quickly
 - **Cache miss:** Data not in cache
 - Fetch data from memory, send it back to processor
 - Retain this data in the cache (replacing some other data)
 - Processor must deal with variable memory access time

Cache Metrics

Hit Ratio:

$$HR = \frac{hits}{hits + misses} = 1 - MR$$

Miss Ratio:

$$MR = \frac{misses}{hits + misses} = 1 - HR$$

Average Memory Access Time (AMAT):

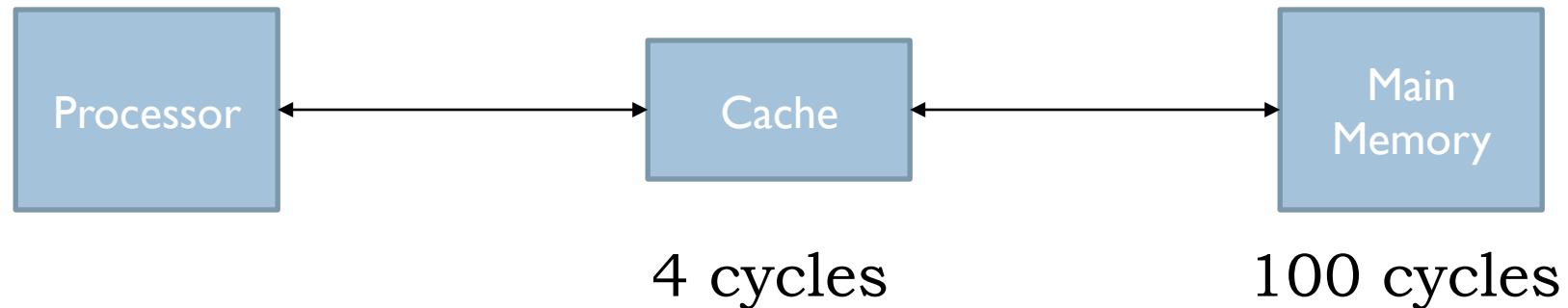
$$AMAT = HitTime + MissRatio \times MissPenalty$$

- Goal of caching is to improve AMAT
- Formula can be applied recursively in multi-level hierarchies:

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times AMAT_{L2} =$$

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times (HitTime_{L2} + MissRatio_{L2} \times AMAT_{L3}) = \dots$$

Example: How High of a Hit Ratio?



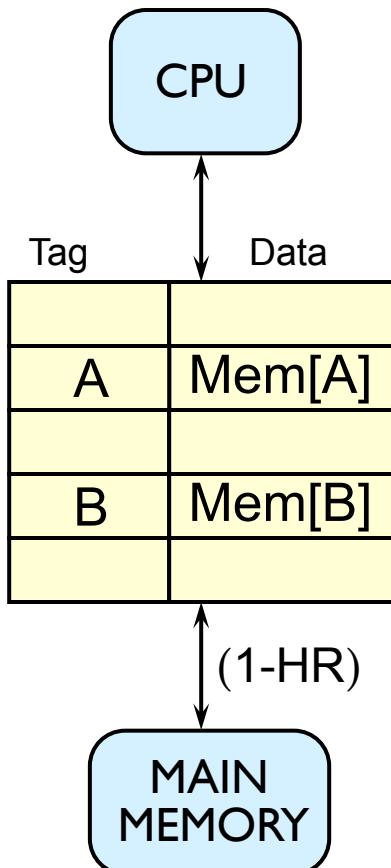
What hit ratio do we need to break even?
(Main memory only: AMAT = 100)

$$100 = 4 + (1 - \text{HR}) \times 100 \Rightarrow \text{HR} = 4\%$$

What hit ratio do we need to achieve AMAT = 5 cycles?

$$5 = 4 + (1 - \text{HR}) \times 100 \Rightarrow \text{HR} = 99\%$$

Basic Cache Algorithm



ON REFERENCE TO Mem[X]:

Look for X among cache tags...

HIT: $X = \text{TAG}(i)$, for some cache line i

- READ: return DATA(i)
- WRITE: change DATA(i); Start Write to Mem(X)

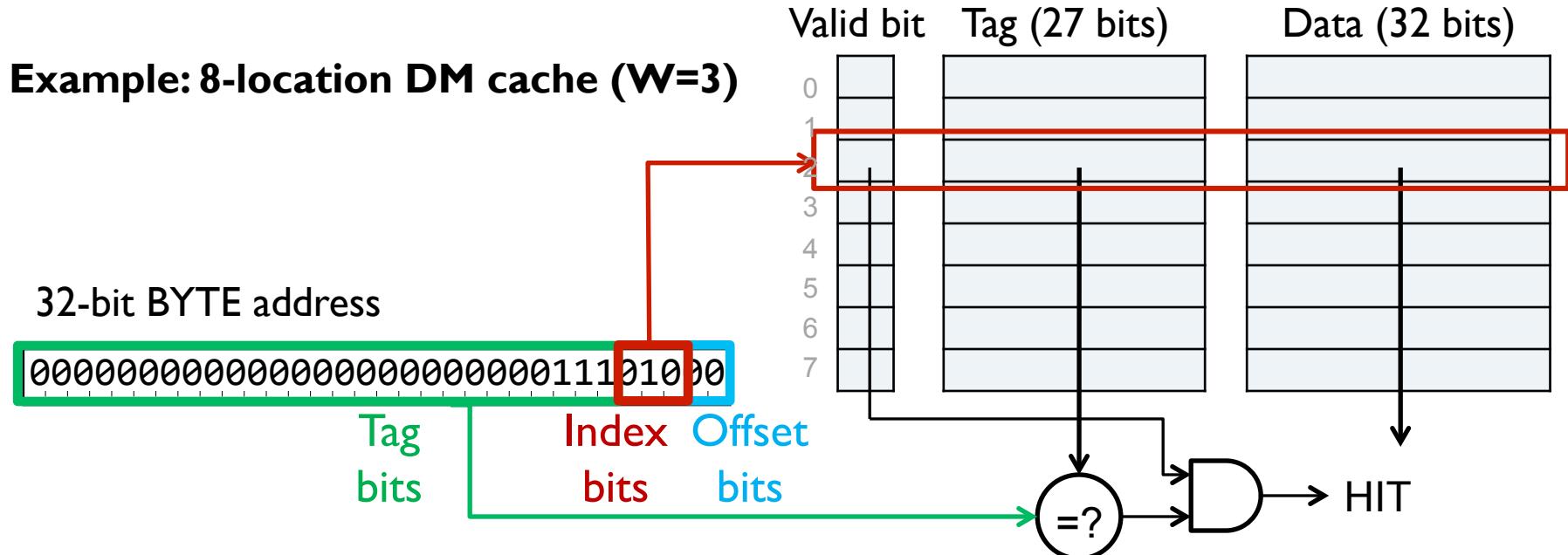
MISS: X not found in TAG of any cache line

- REPLACEMENT SELECTION:
Select some line k to hold Mem[X] (Allocation)
- READ: Read Mem[X]
Set TAG(k)=X, DATA(k)=Mem[X]
- WRITE: Start Write to Mem(X)
Set TAG(k)=X, DATA(k)= new Mem[X]

Q: How do we “search” the cache?

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



Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → **6 index bits**

Read Mem[0x400C]

0100 0000 0000 1100
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?

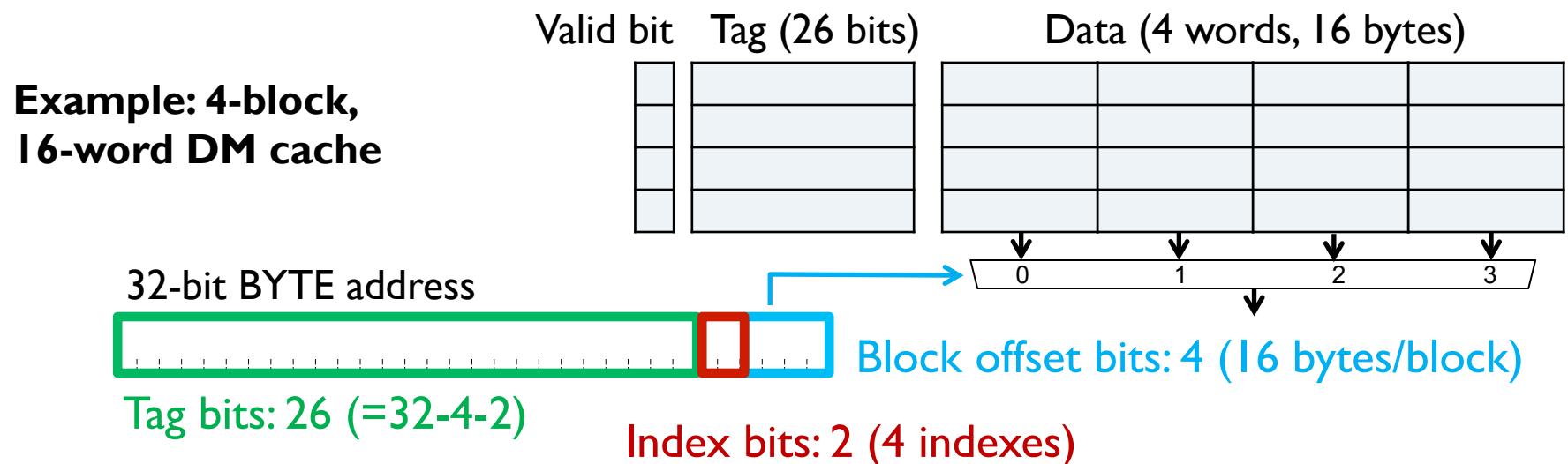
TAG: 0x58 → 0101 1000 iii i00 (substitute line # for iiiii) → 0x5800, 0x5804, 0x5808

Part of the address (index bits) is **encoded in the location!**
Tag + Index bits unambiguously identify the data's address

Block Size

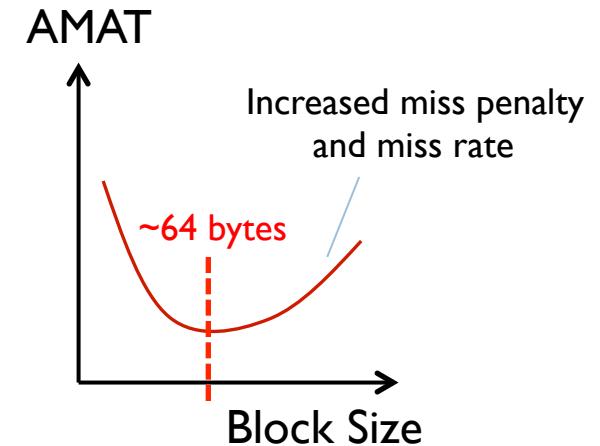
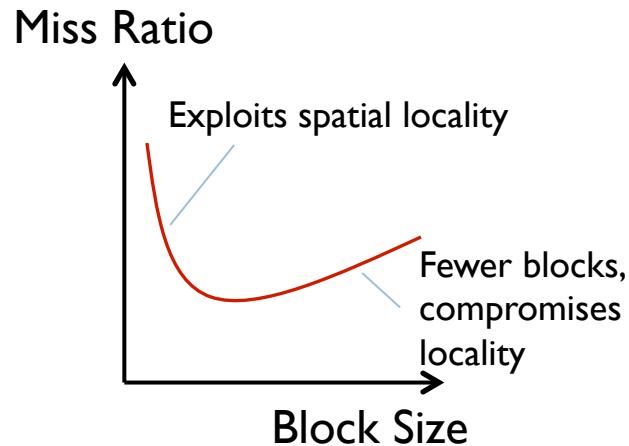
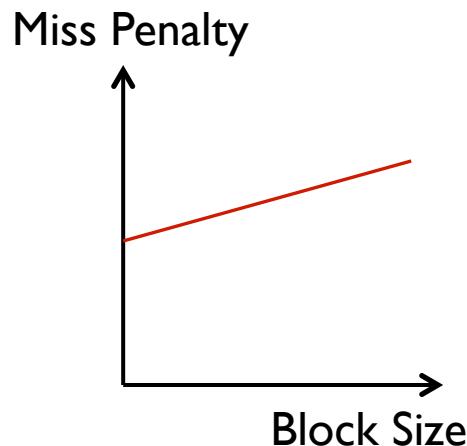
Take advantage of locality: increase block size

- Another advantage: Reduces size of tag memory!
- Potential disadvantage: Fewer blocks in the cache



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



Direct-Mapped Cache Problem: Conflict Misses

Loop A:
Pgm at
1024,
data at
37:

Word Address	Cache Line index	Hit/ Miss
1024	0	HIT
37	37	HIT
1025	1	HIT
38	38	HIT
1026	2	HIT
39	39	HIT
1024	0	HIT
37	37	HIT
...		

Assume:
1024-line DM cache
Block size = 1 word
Consider looping code, in
steady state
Assume WORD, not BYTE,
addressing

Loop B:
Pgm at
1024,
data at
2048:

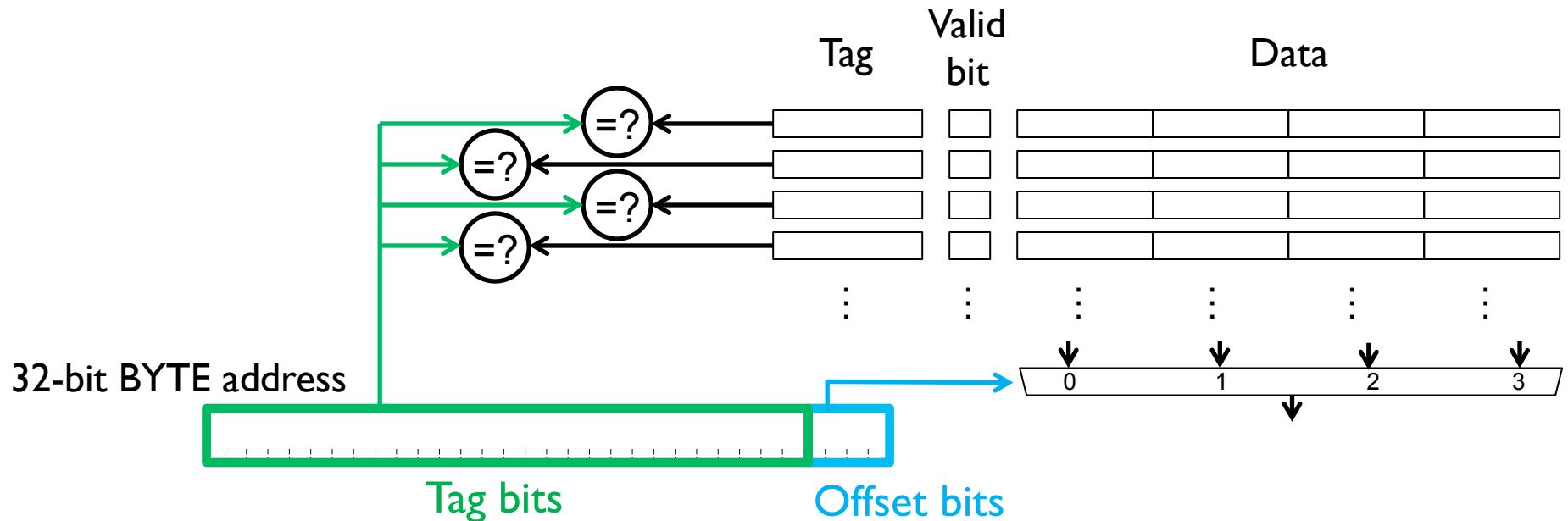
1024	0	MISS
2048	0	MISS
1025	1	MISS
2049	1	MISS
1026	2	MISS
2050	2	MISS
1024	0	MISS
2048	0	MISS
...		

Inflexible mapping (each
address can only be in one
cache location) → **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)



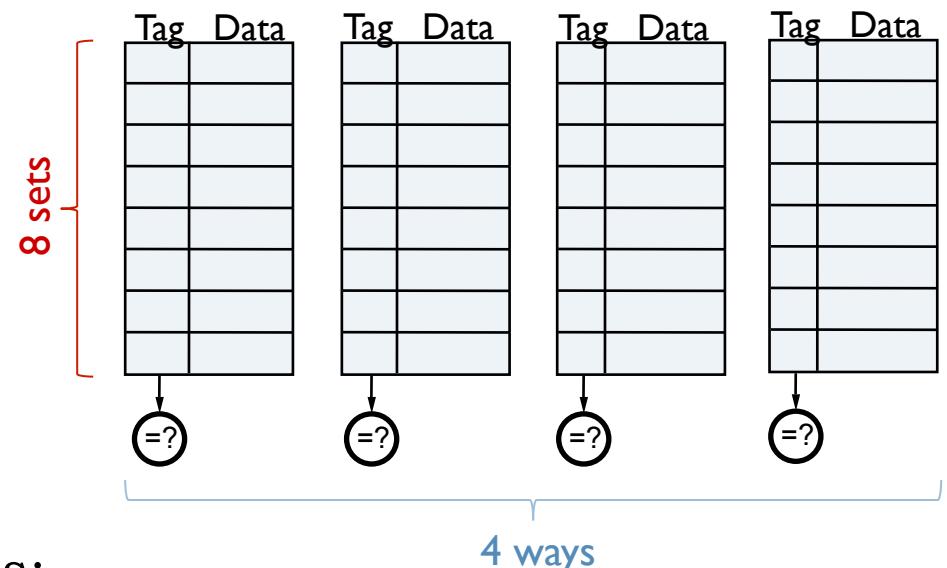
N-way Set-Associative Cache

- 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

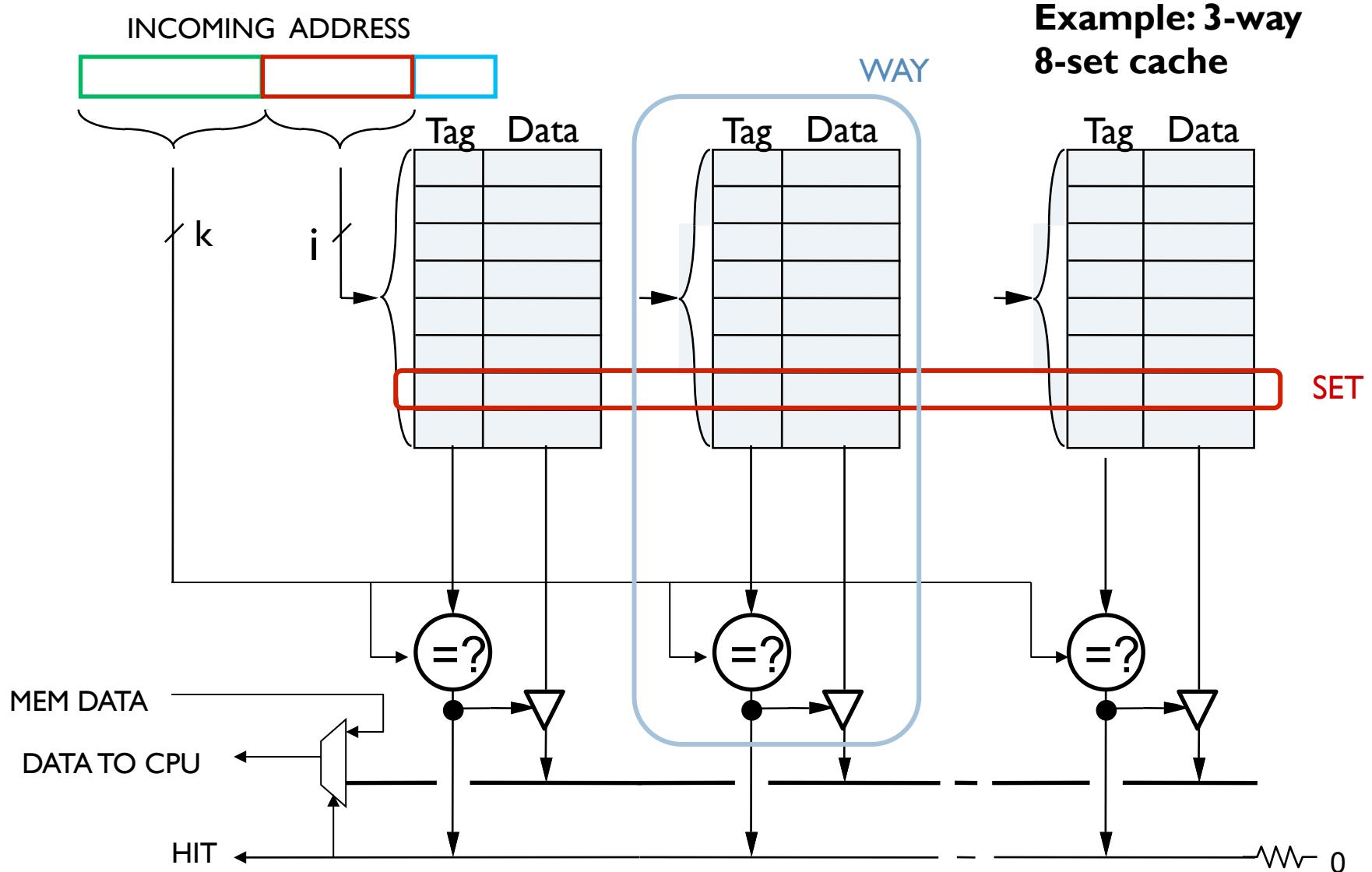


- An N-way cache can be seen as:

- N direct-mapped caches in parallel

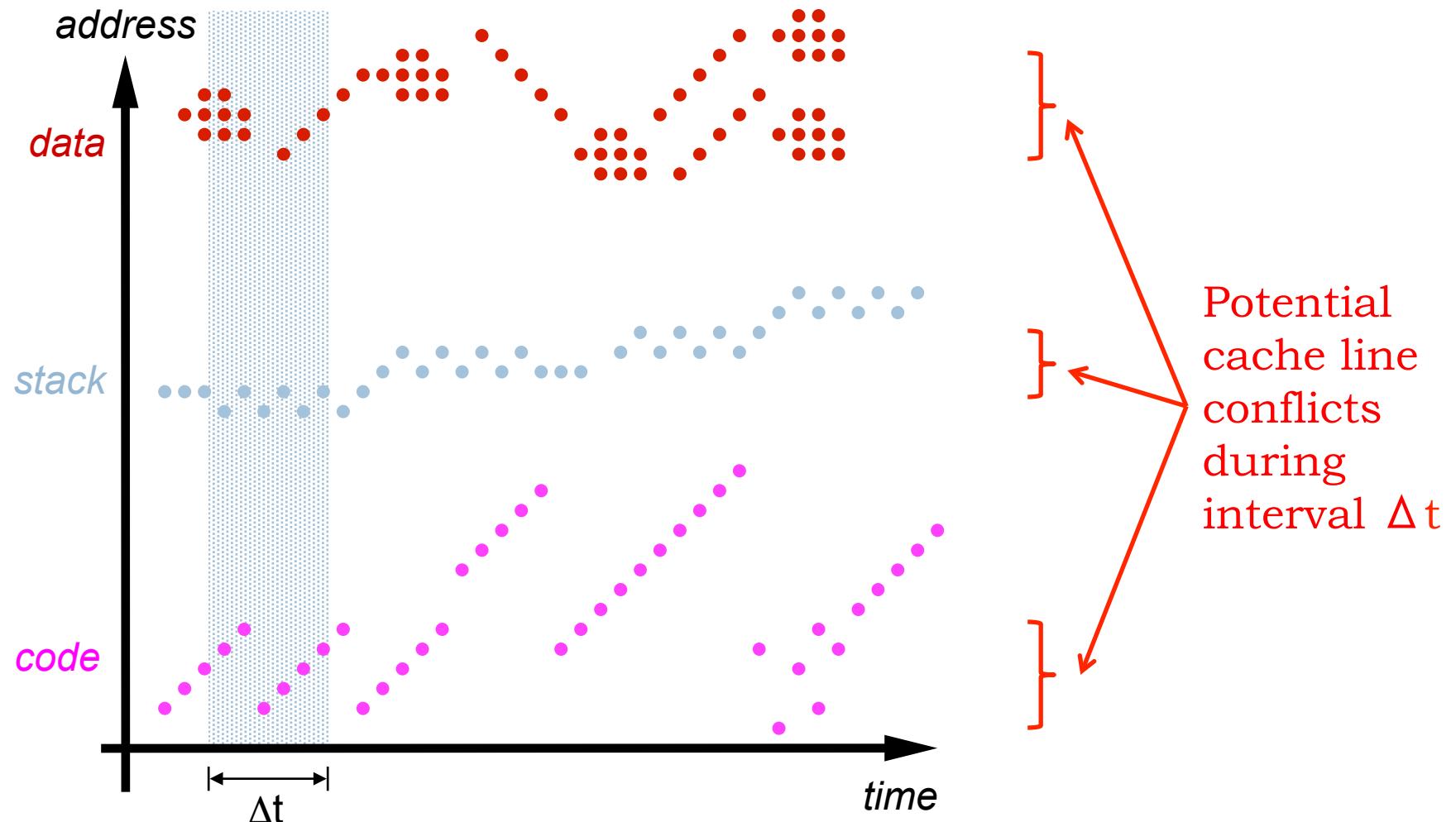
- Direct-mapped and fully-associative are just special cases of N-way set-associative

N-way Set-Associative Cache



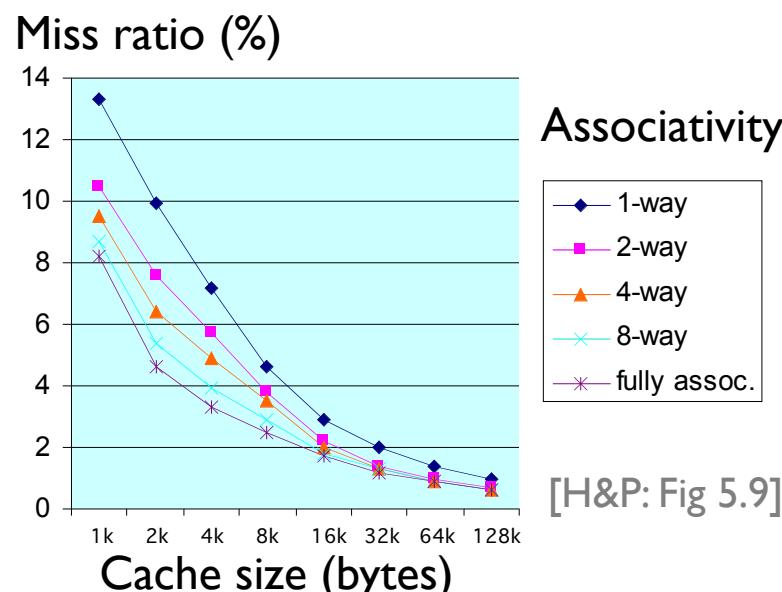
“Let me count the ways.”

Elizabeth Barrett Browning



Associativity Tradeoffs

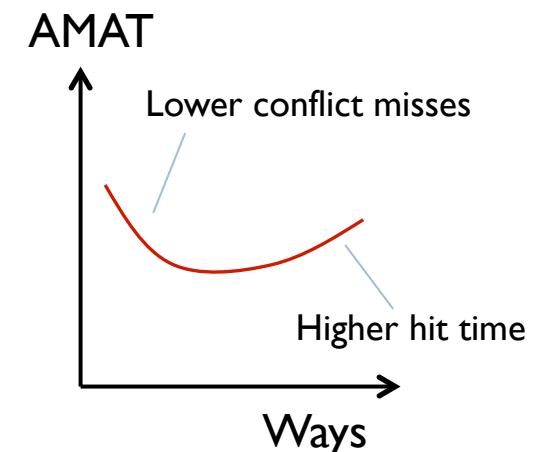
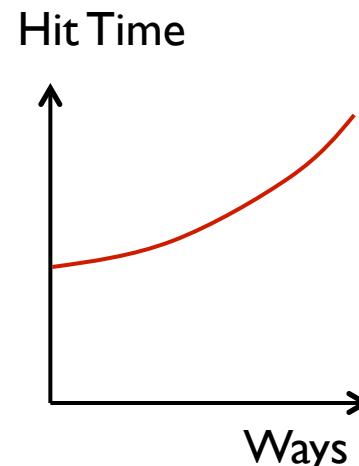
- More ways...
 - Reduce conflict misses
 - Increase hit time



[H&P: Fig 5.9]

Little additional benefits
beyond 4 to 8 ways

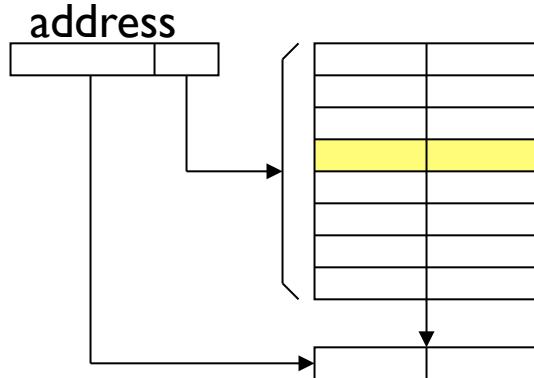
$$AMAT = HitTime + MissRatio \times MissPenalty$$



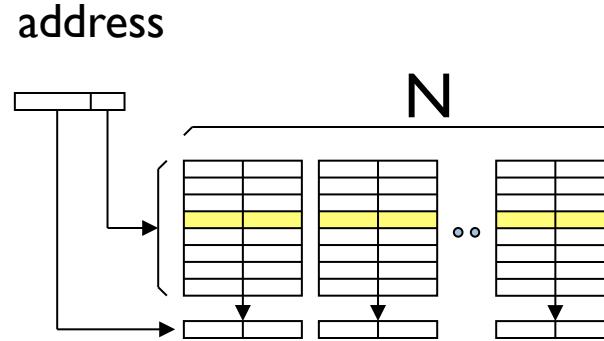
Associativity Implies Choices

Issue: Replacement Policy

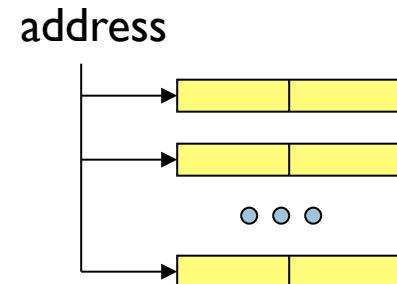
Direct-mapped



N-way set-associative



Fully associative



- Compare addr with only one tag
- Location A can be stored in exactly one cache line

- Compare addr with N tags simultaneously
- Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

- Compare addr with each tag simultaneously
- Location A can be stored in any cache line

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

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”

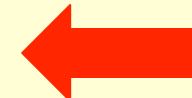
- Fastest, low bandwidth, more complex
- Commonly implemented in current systems

Write-Back

ON REFERENCE TO $\text{Mem}[X]$: Look for 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 $\text{Mem}[X]$~~



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 $\text{Mem}[X]$
 - Write Back: Write $\text{Data}[k]$ to $\text{Mem}[\text{Address from Tag}[k]]$



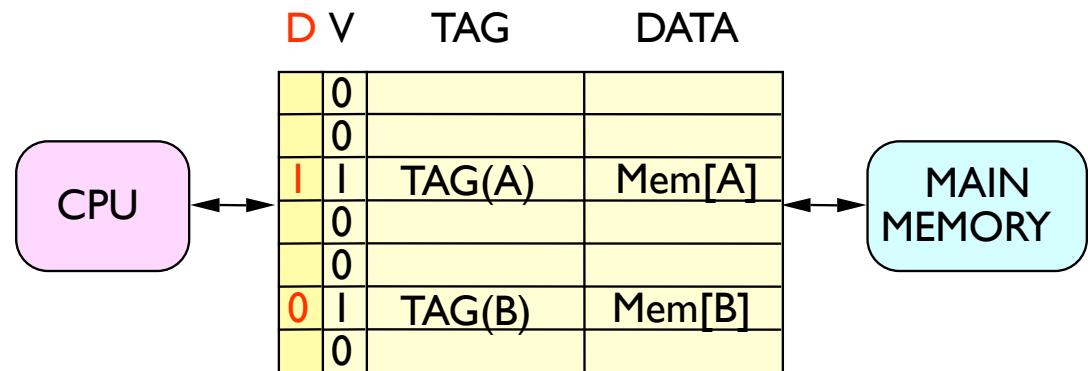
- READ: Read $\text{Mem}[X]$
 - Set $\text{Tag}[k] = \text{TAG}(X)$, $\text{Data}[k] = \text{Mem}[X]$

- WRITE: ~~Start Write to $\text{Mem}[X]$~~
 - Set $\text{Tag}[k] = \text{TAG}(X)$, $\text{Data}[k] = \text{new Mem}[X]$



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...

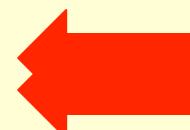
HIT: $\text{TAG}(X) == \text{Tag}[i]$, for some cache block i

- READ: return $\text{Data}[i]$
- WRITE: change $\text{Data}[i]$ ~~Start Write to $\text{Mem}[X]$~~ $D[i]=1$



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 $D[k] == 1$ (Writeback) Write $\text{Data}[k]$ to $\text{Mem}[\text{Address of Tag}[k]]$
- READ: Read $\text{Mem}[X]$; Set $\text{Tag}[k] = \text{TAG}(X)$, $\text{Data}[k] = \text{Mem}[X]$, $D[k]=0$
- WRITE: ~~Start Write to $\text{Mem}[X]$~~ $D[k]=1$
 - Set $\text{Tag}[k] = \text{TAG}(X)$, $\text{Data}[k] = \text{new Mem}[X]$



Summary: Cache Tradeoffs

$$AMAT = HitTime + MissRatio \times MissPenalty$$

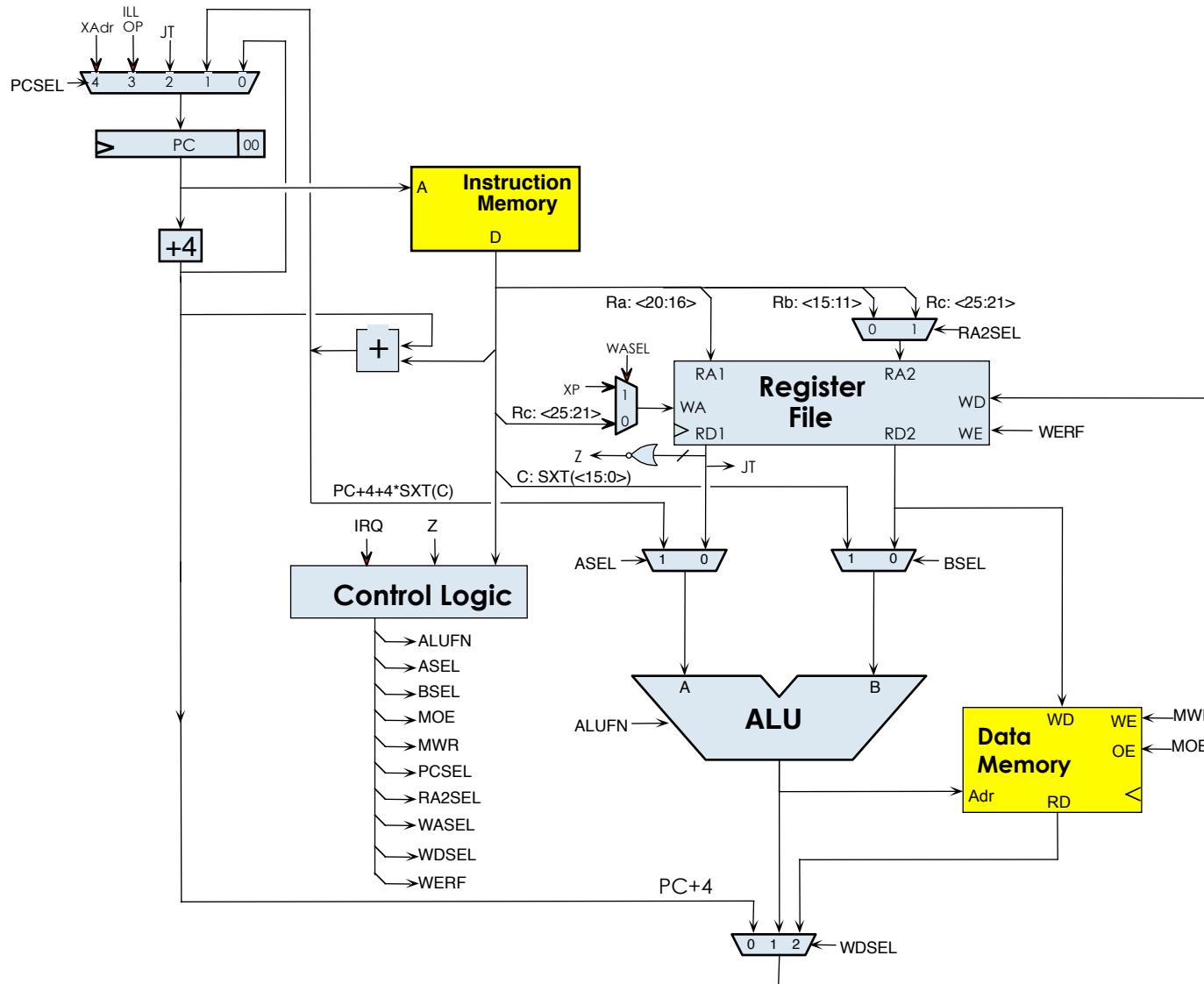
- Larger **cache size**: Lower miss rate, higher hit time
- Larger **block size**: Trade off spatial for temporal locality, higher miss penalty
- More **associativity** (ways): Lower miss rate, higher hit time
- More intelligent **replacement**: Lower miss rate, higher cost
- **Write policy**: Lower bandwidth, more complexity
- How to navigate all these dimensions? Simulate different cache organizations on real programs

15. Pipelining the Beta

6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

Reminder: Single-Cycle Beta



Single-Cycle Beta Performance

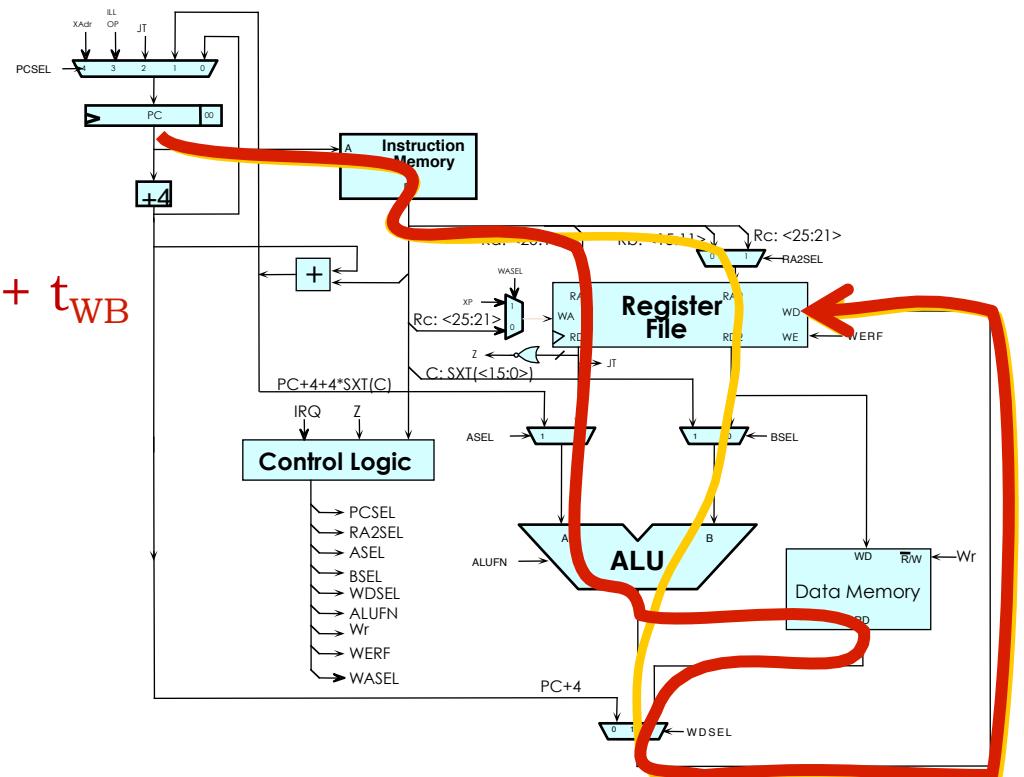
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

CPI t_{CLK}

- CPI = 1
- t_{CLK} = Longest path for any instruction

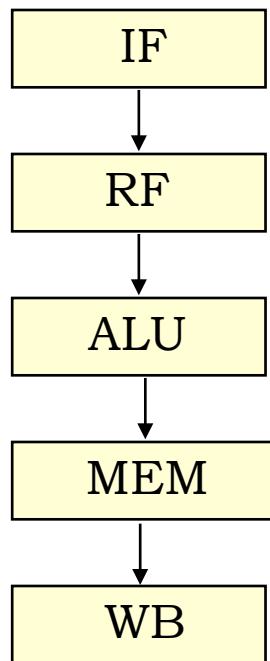
$$t_{\text{CLK}} \approx t_{\text{IFETCH}} + t_{\text{RF}} + t_{\text{ALU}} + t_{\text{MEM}} + t_{\text{WB}}$$

- Slow
- Inflexible: Instructions with smaller critical path cannot execute faster



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

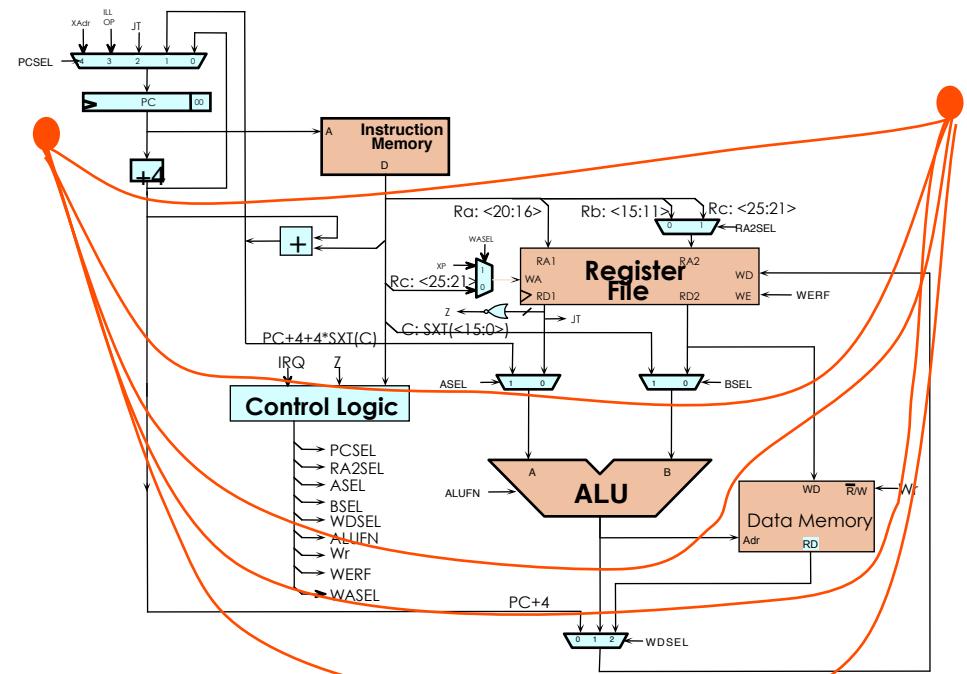
Write-Back stage: writes result back into register file.

$$t_{CLK} = \max\{t_{IFETCH}, t_{RF}, t_{ALU}, t_{MEM}, t_{WB}\}$$

Why isn't this a 20-minute lecture?

We know how to pipeline combinational circuits,
what's the big deal?

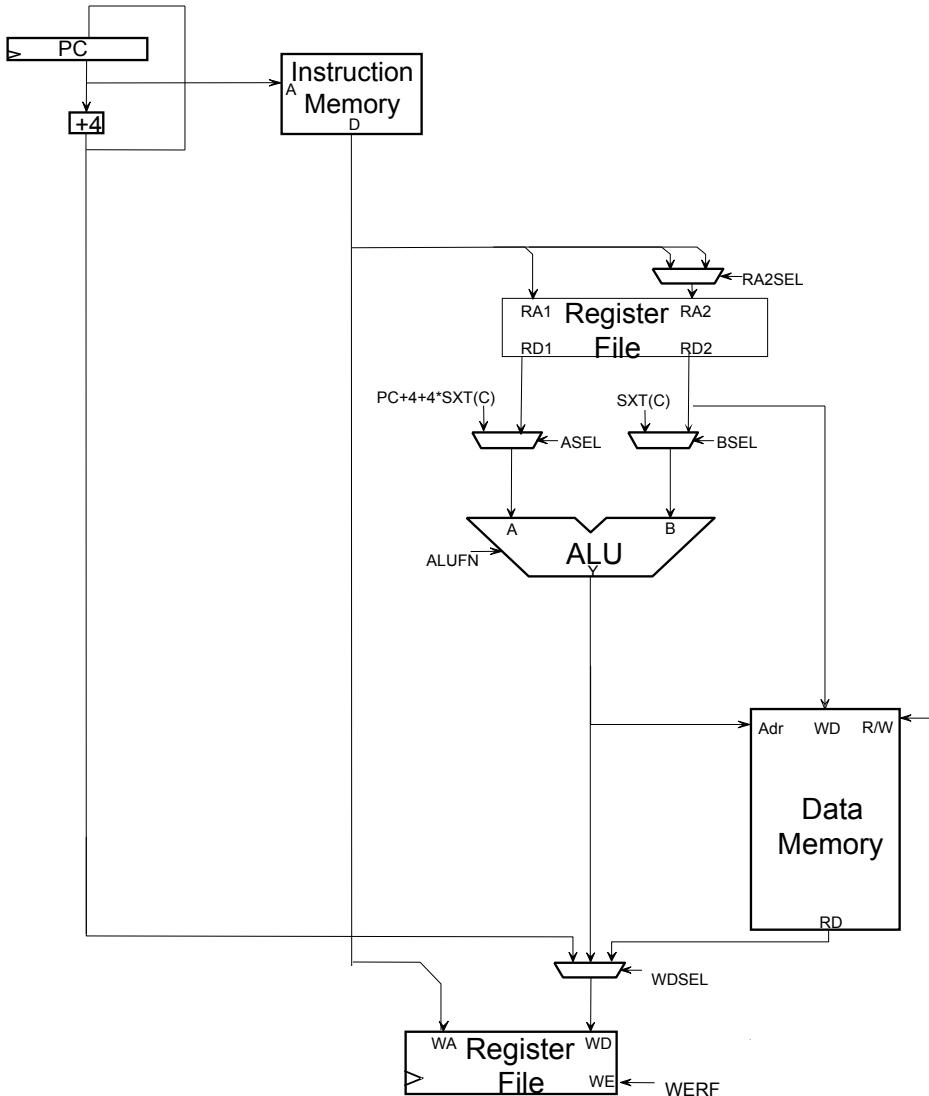
- Beta has state: PC, Register file, Memories
- There are dependencies we cannot break!
 - To compute the next PC
 - To write result into the register file
- We'll be addressing these issues as we examine the operation of our execution pipeline.



Pipeline Hazards

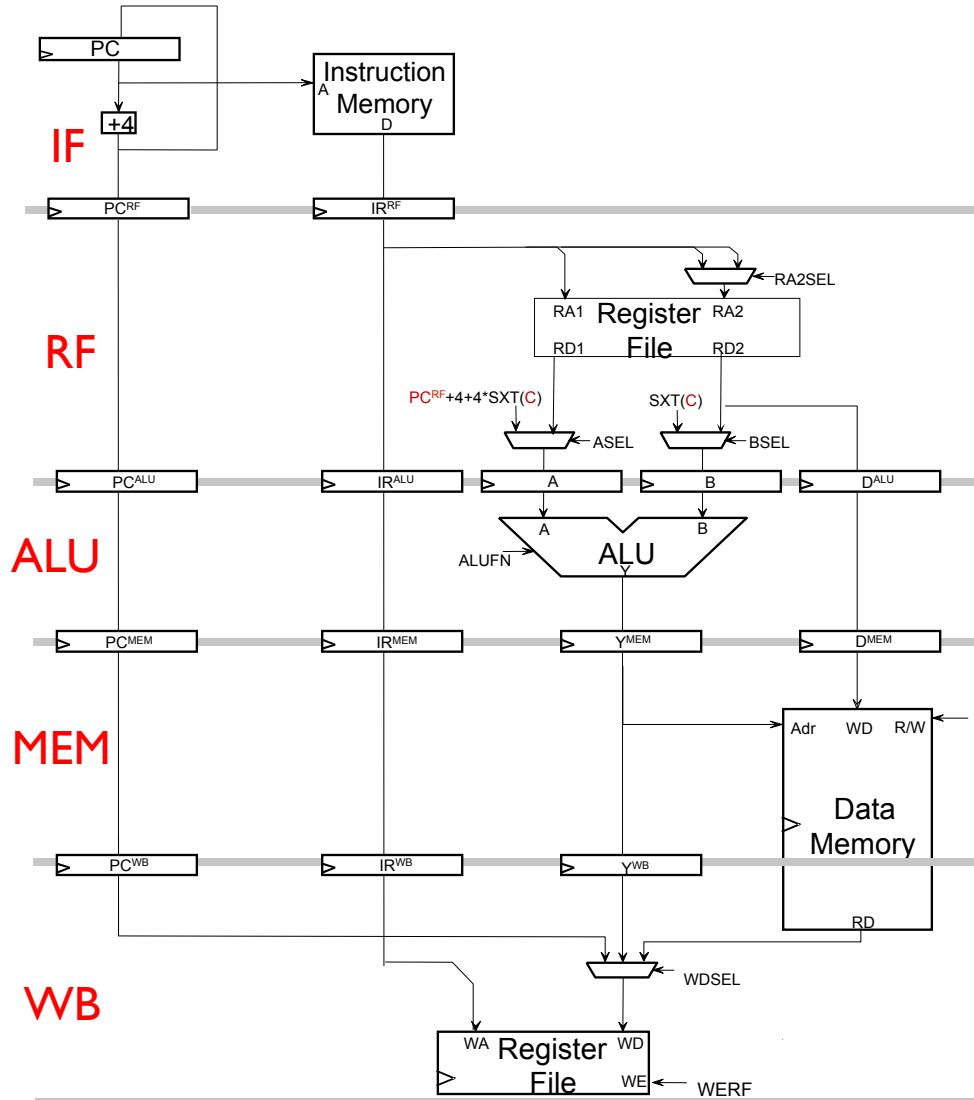
- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
 - A data value → Data hazard
 - The program counter → Control hazard
(branches, jumps, exceptions)
- Plan of attack:
 1. Design a 5-stage pipeline that works with sequences of independent instructions
 2. Handle data hazards
 3. Handle control hazards

Simplified Unpipelined Beta Datapath



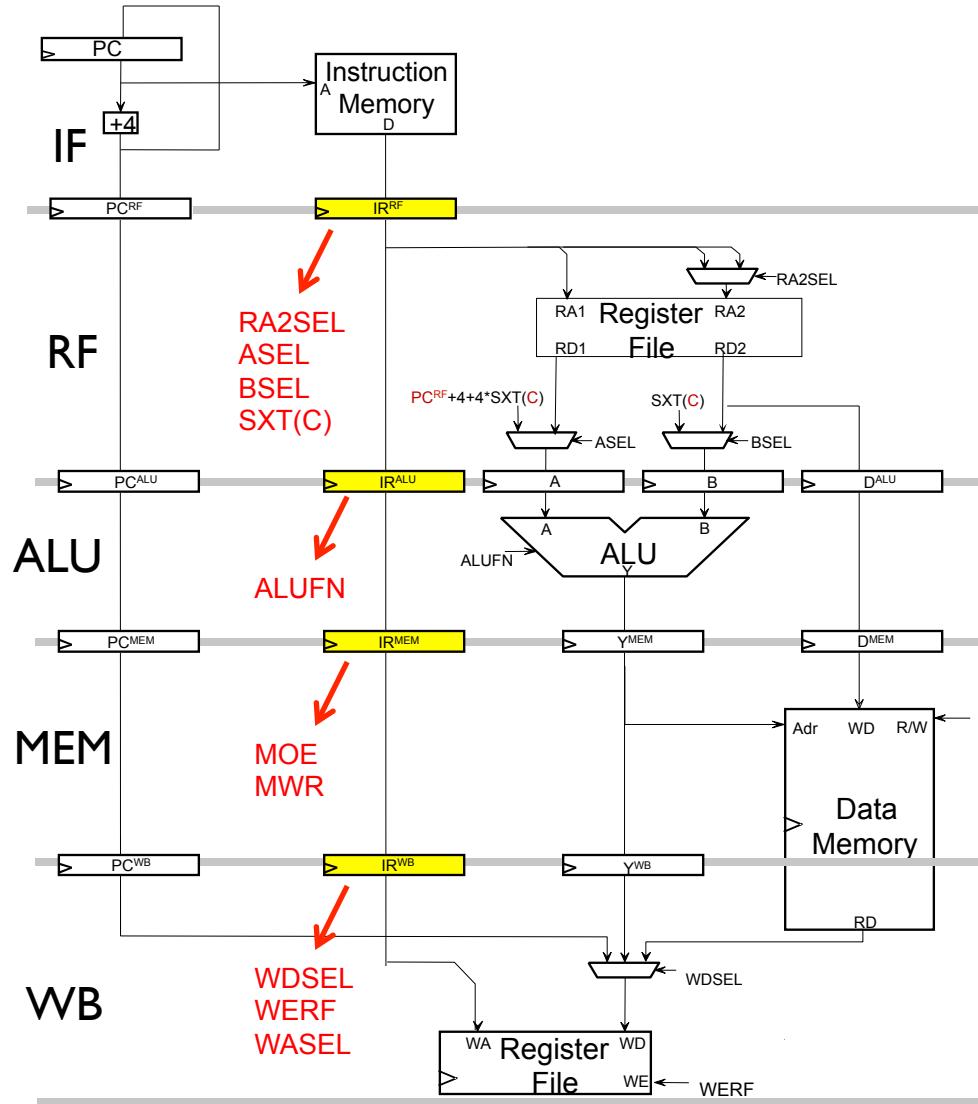
- NextPC = PC+4 (we'll worry about control hazards later)
 - Same register file appears twice in the diagram
 - Top: reads
 - Bottom: writes

5-Stage Pipelined Datapath



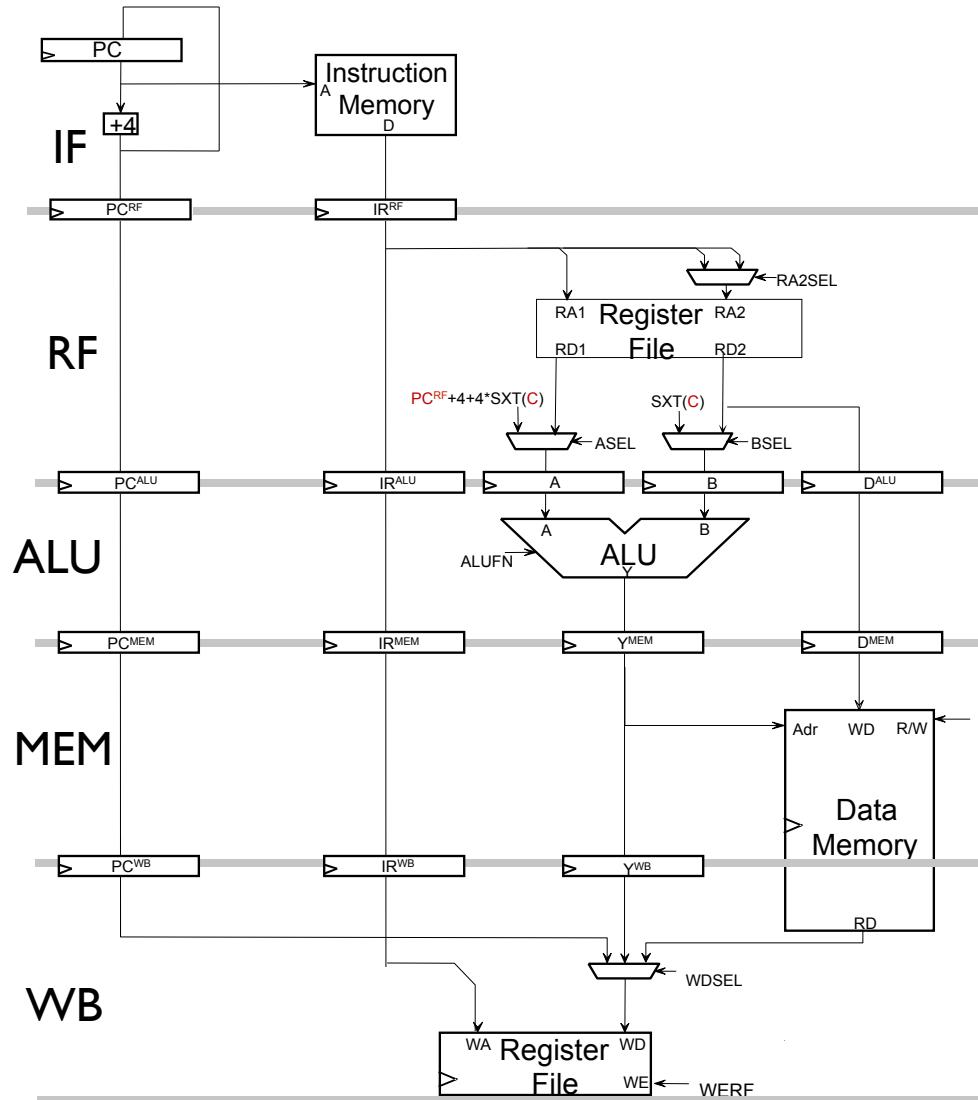
- Pipeline registers separate different stages:
 - IF – instruction fetch
 - RF – register file access
 - ALU – compute result
 - MEM – memory access
 - WB – write back to reg. file
- Each stage services one instruction per cycle
- Data memory reads are now pipelined, not combinational
 - Data read appears in RD the next cycle

Pipelined Control



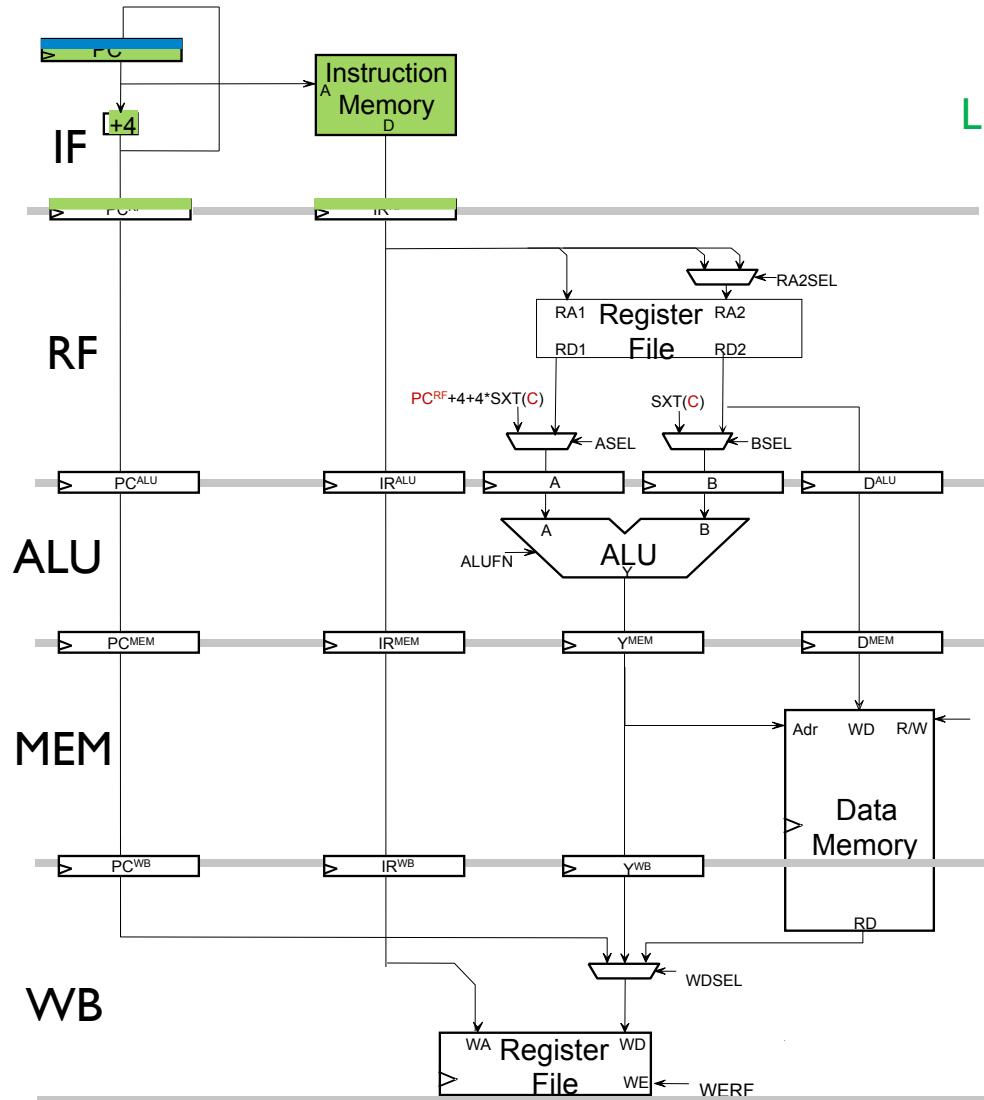
- Instruction contents propagated through the pipeline in Instruction Registers (IR^{RF}, IR^{ALU}, ...)
- Control signals for each stage generated from corresponding IR
 - e.g. ASEL uses IR^{RF} opcode, WERF uses IR^{WB}, etc
- Pipeline hazards will require new control signals

Pipelined Execution Example



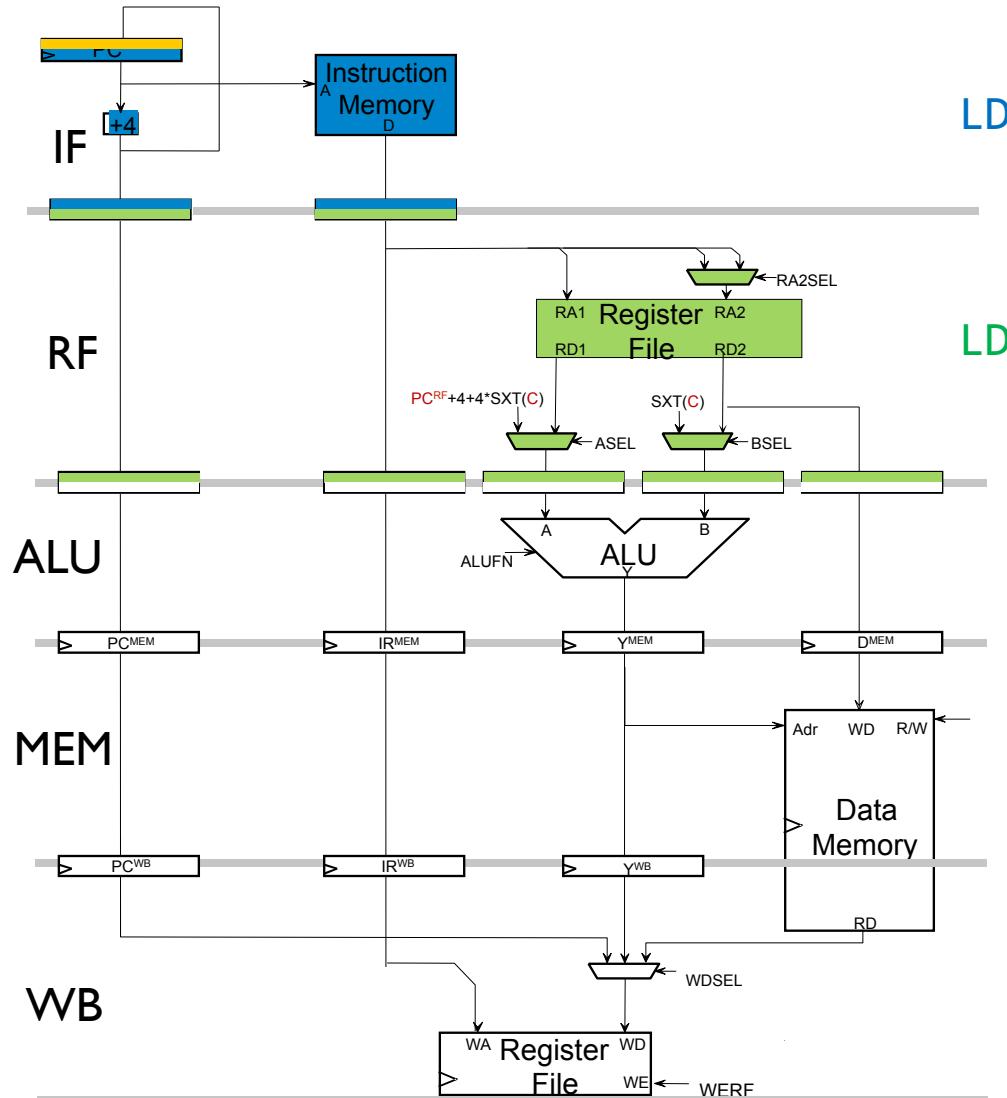
Sequence of instructions
without data or control
dependences

Example: Cycle 1



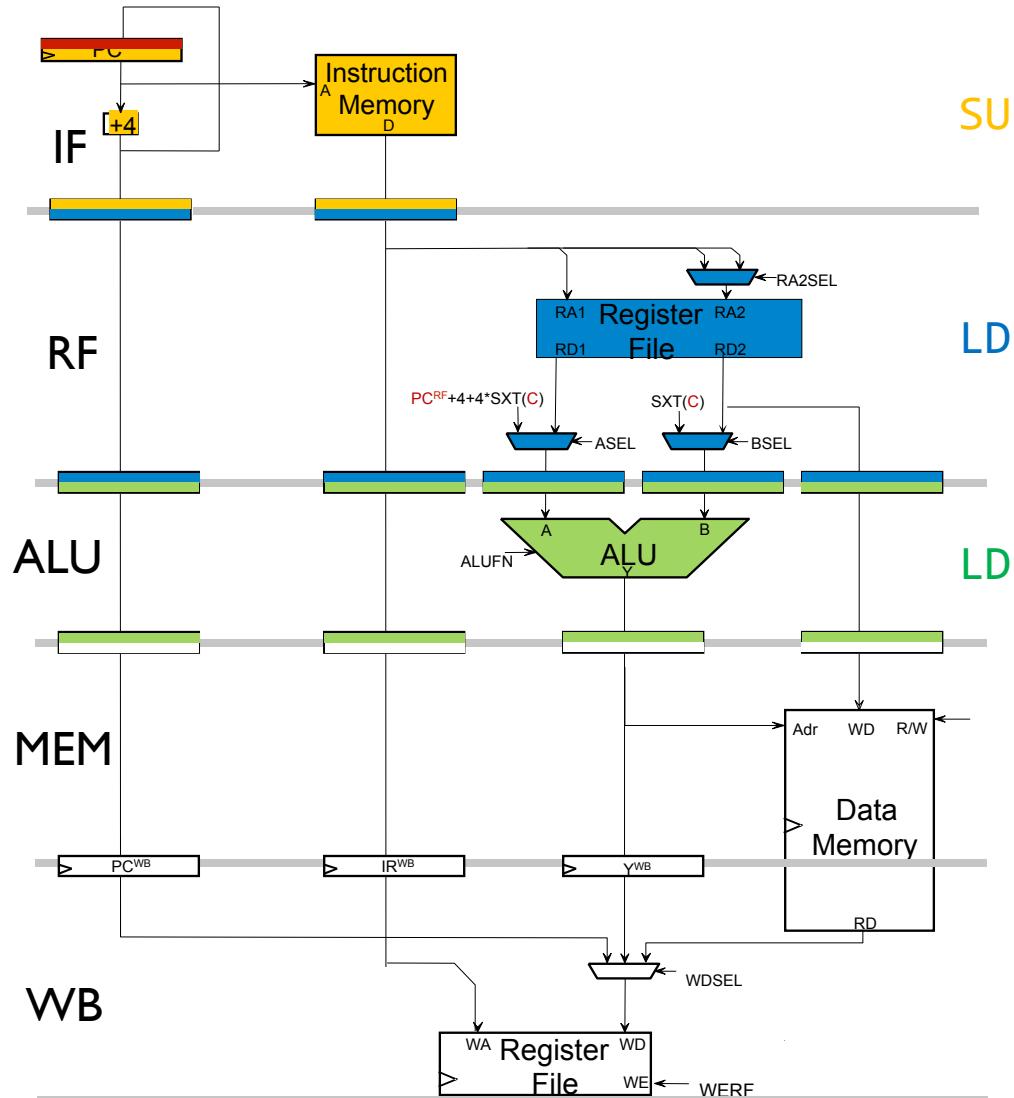
LD(R1, 4, R2)
 LD(R3, 8, R4)
 SUB(R6, R7, R8)
 XOR(R9, R10, R11)
 MUL(R12, R13, R14)
 ADD(R15, 1, R16)

Example: Cycle 2



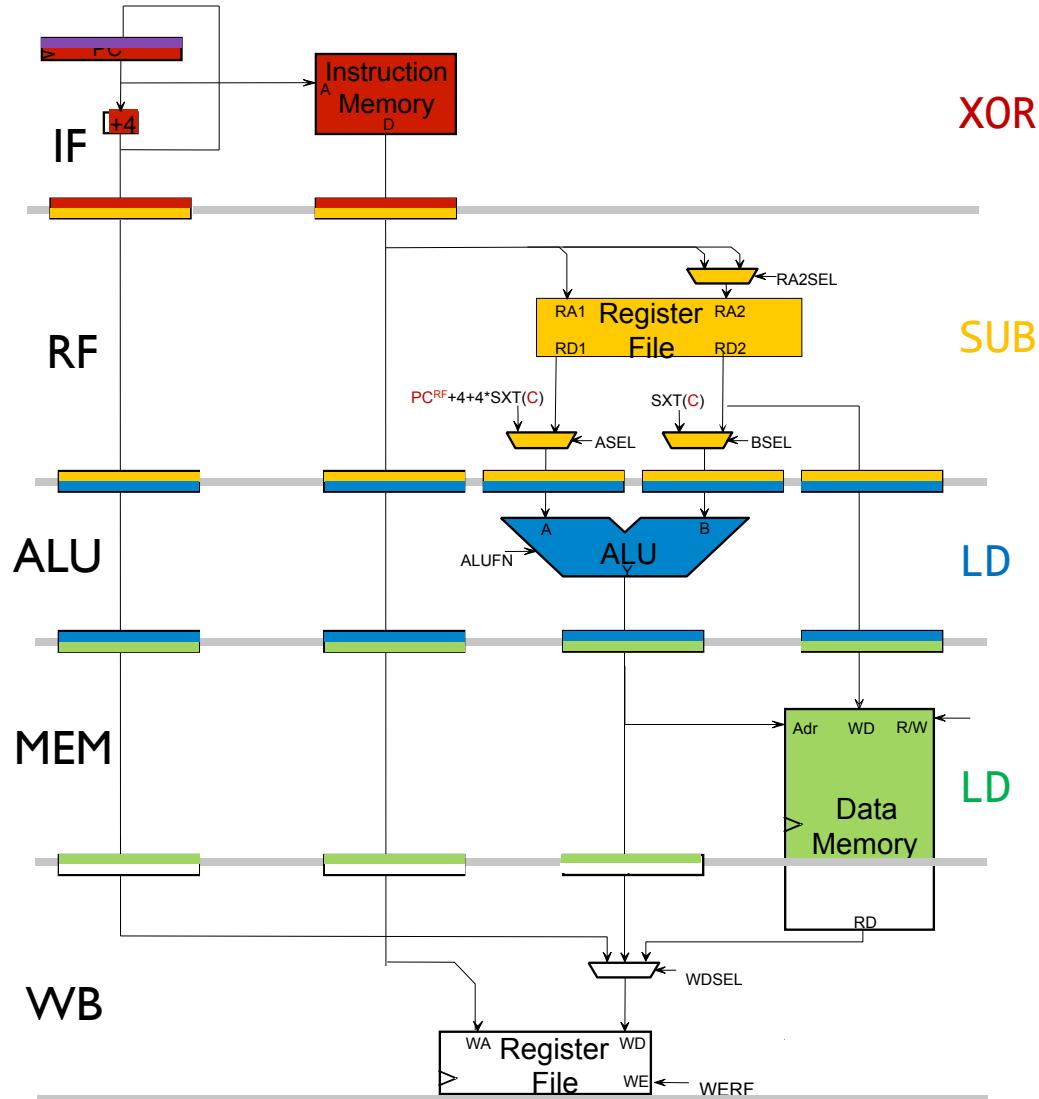
LD(R1, 4, R2)
 LD(R3, 8, R4)
 SUB(R6, R7, R8)
 XOR(R9, R10, R11)
 MUL(R12, R13, R14)
 ADD(R15, 1, R16)

Example: Cycle 3



LD(R1, 4, R2)
 LD(R3, 8, R4)
 SUB(R6, R7, R8)
 XOR(R9, R10, R11)
 MUL(R12, R13, R14)
 ADD(R15, 1, R16)

Example: Cycle 4



XOR

LD(R1, 4, R2)

LD(R3, 8, R4)

SUB

SUB(R6, R7, R8)

XOR(R9, R10, R11)

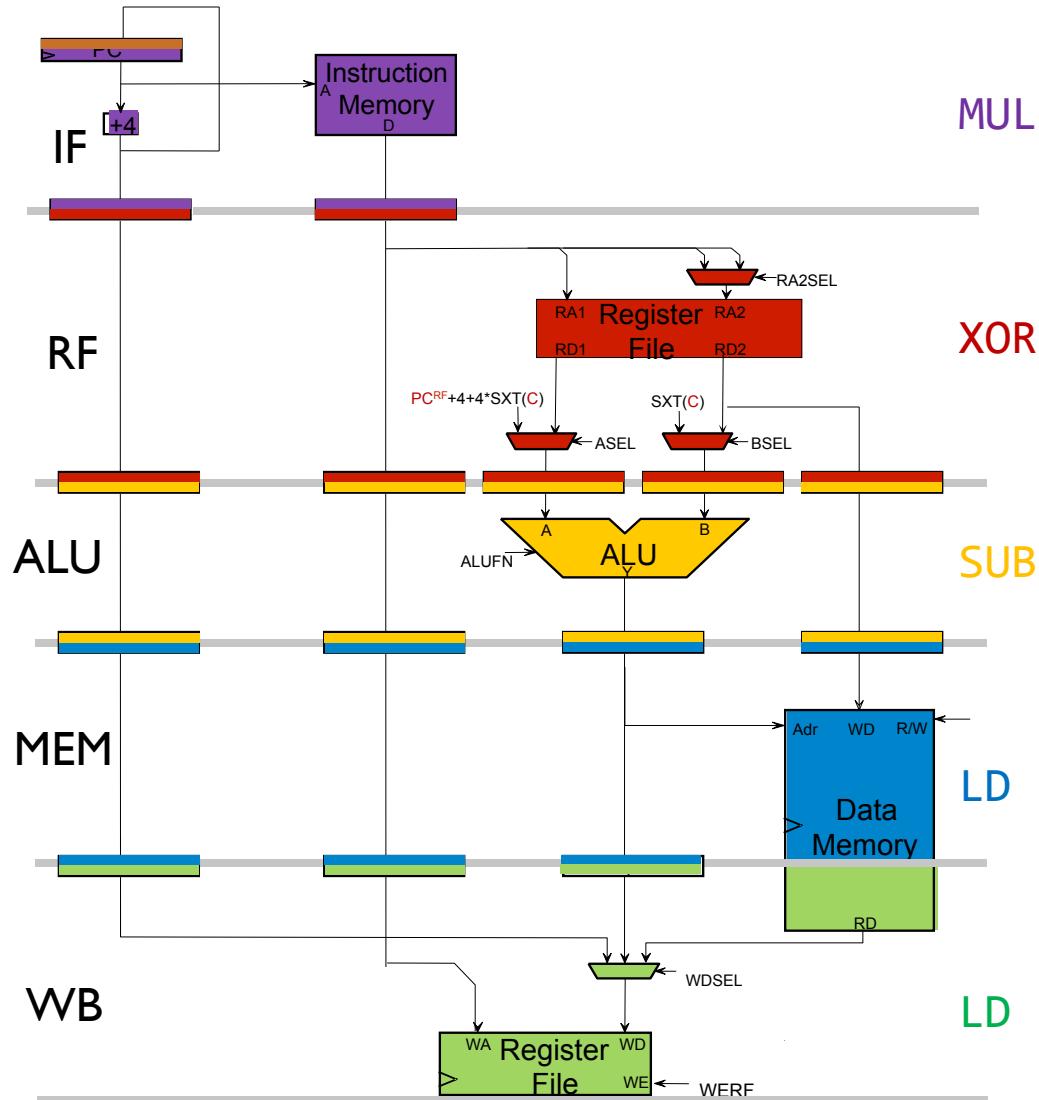
MUL(R12, R13, R14)

ADD(R15, 1, R16)

LD

First LD starts data
memory read
Data not yet available!

Example: Cycle 5



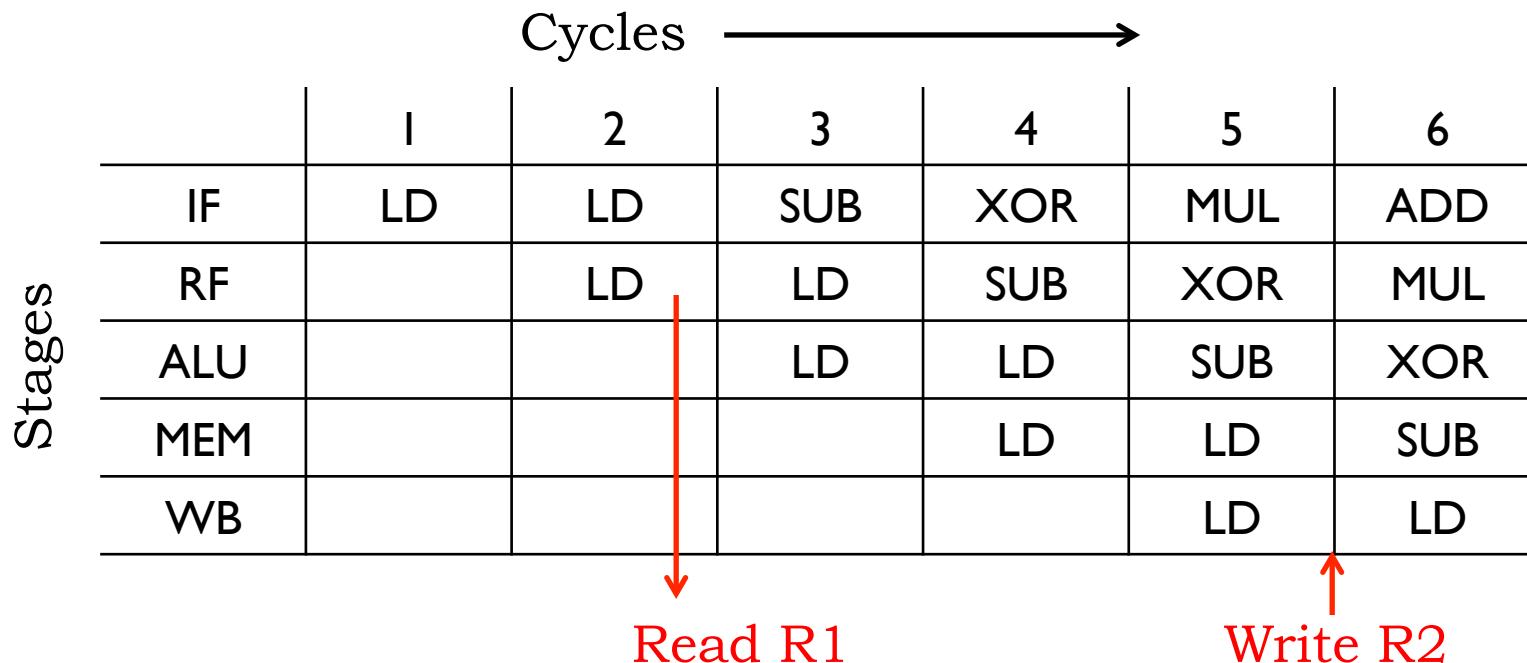
$LD(R1, 4, R2)$
 $LD(R3, 8, R4)$
 $SUB(R6, R7, R8)$
 $XOR(R9, R10, R11)$
 $MUL(R12, R13, R14)$
 $ADD(R15, 1, R16)$

Second LD starts data memory read

Data for first LD available in RD

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)



Data Hazards

- Consider this instruction sequence:

ADDC(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)

	I	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	

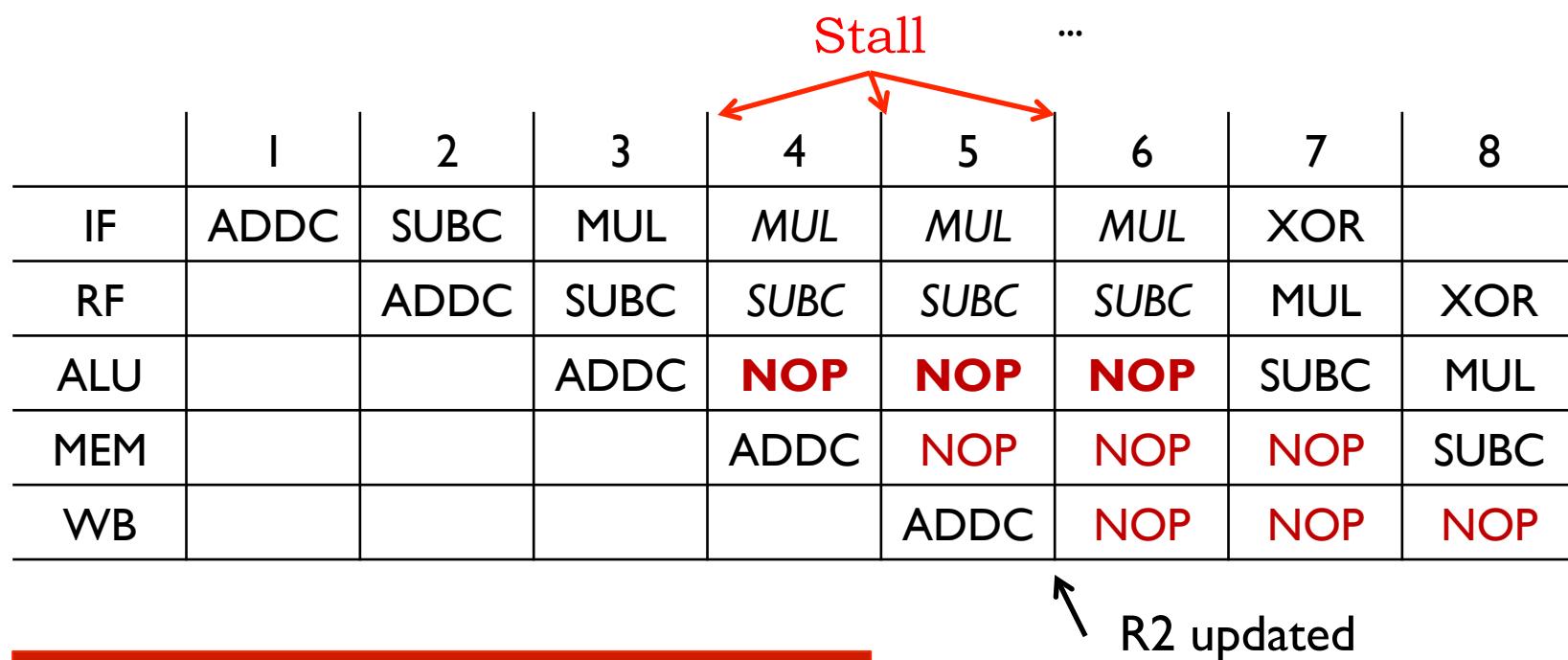
- SUBC reads R2 on cycle 3, but ADDC does not update it until end of cycle 5 → **R2 is stale!**
- Pipeline must maintain correct behavior...**

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

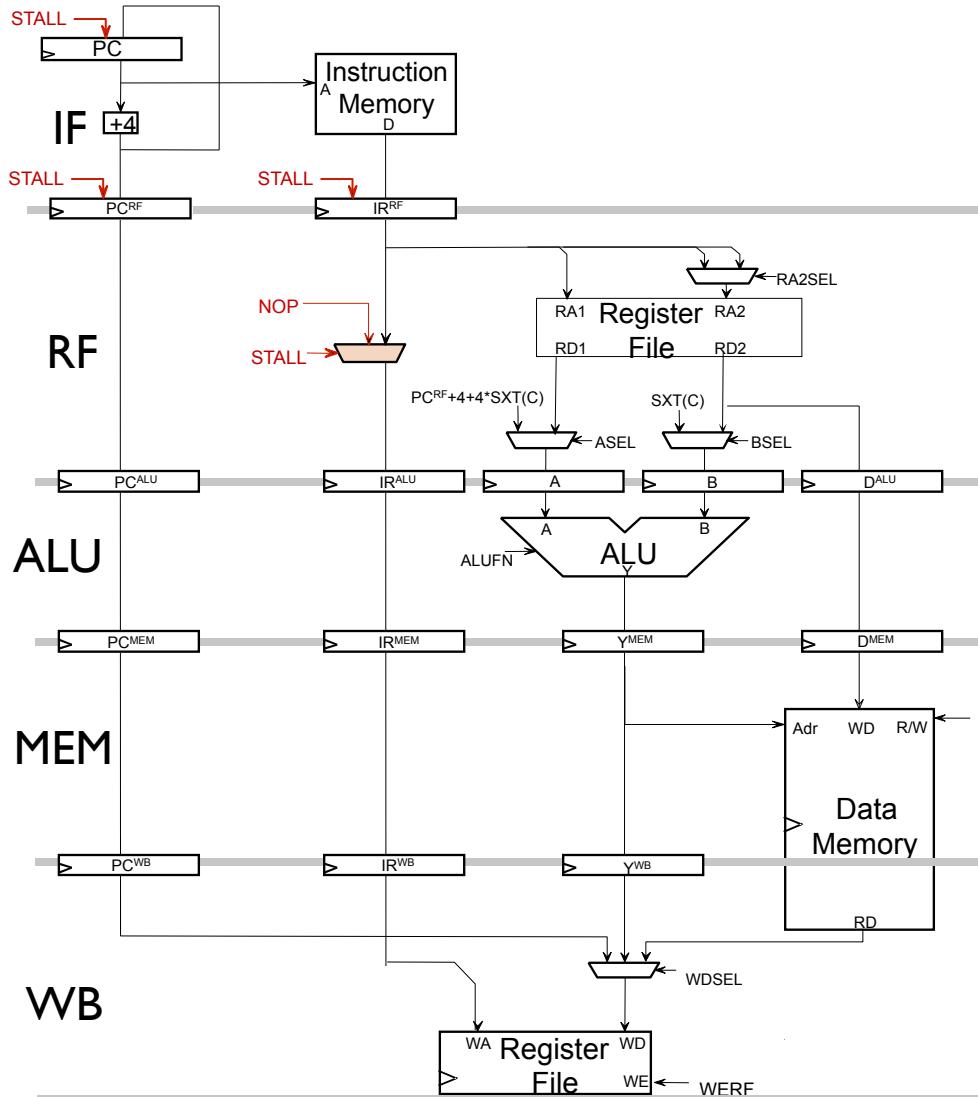
Resolving Data Hazards (1)

- 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)
...



Stalls increase CPI!

Stall Logic



- New **STALL** control signal
- **STALL==1**
 - Disables PC and RF pipeline registers
 - Injects NOP instruction into ALU stage
- NOP = No-operation, e.g., ADD(R31, R31, R31)
- Control logic sets **STALL=1** if source registers of instruction in RF match destination register on ALU, MEM, or WB *(except when source is R31)*

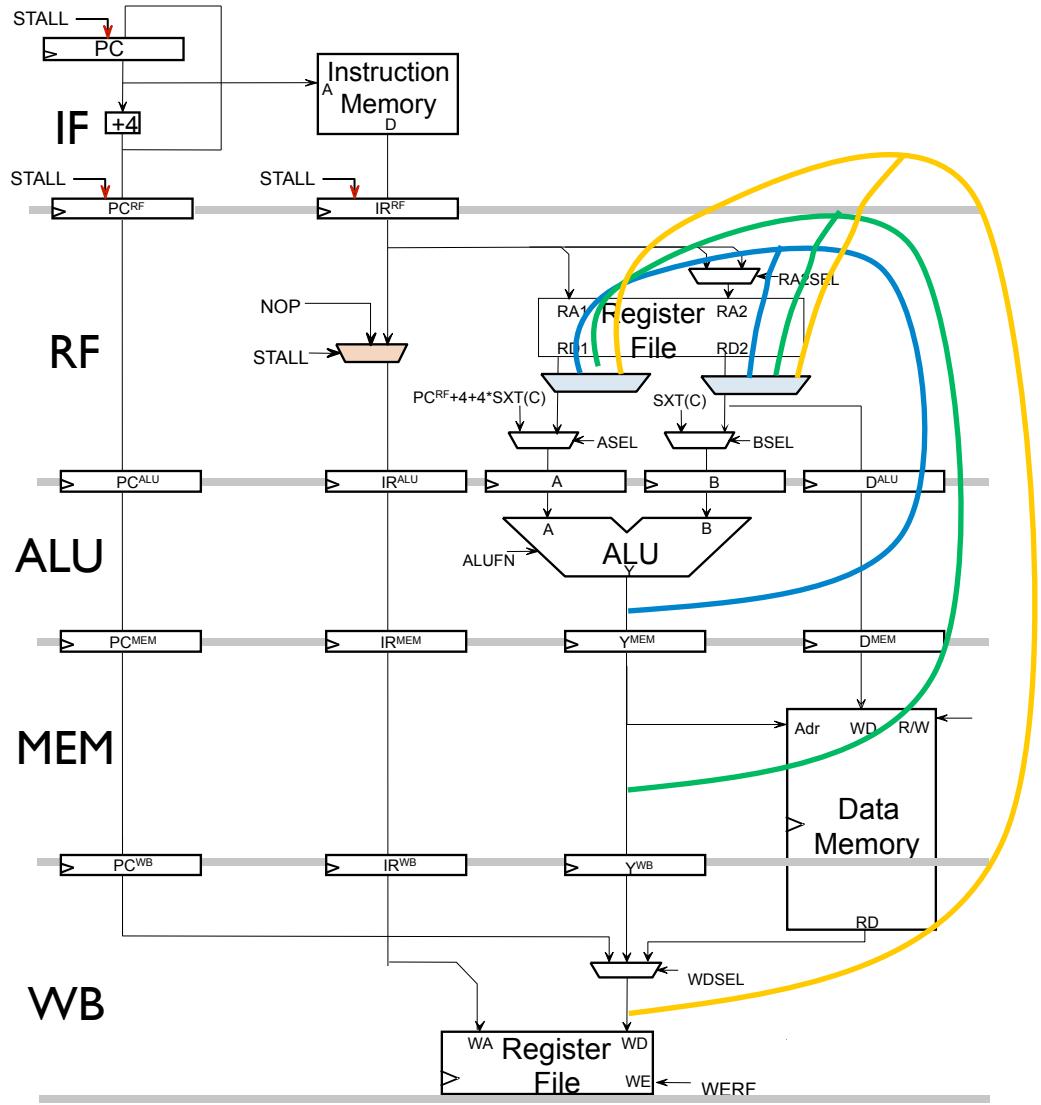
Resolving Data Hazards (2)

- 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!

	I	2	3	4	5
IF	ADDC	SUBC	MUL	XOR	
RF		ADDC	SUBC	MUL	XOR
ALU			ADDC	SUBC	MUL
MEM				ADDC	SUBC
WB					ADDC

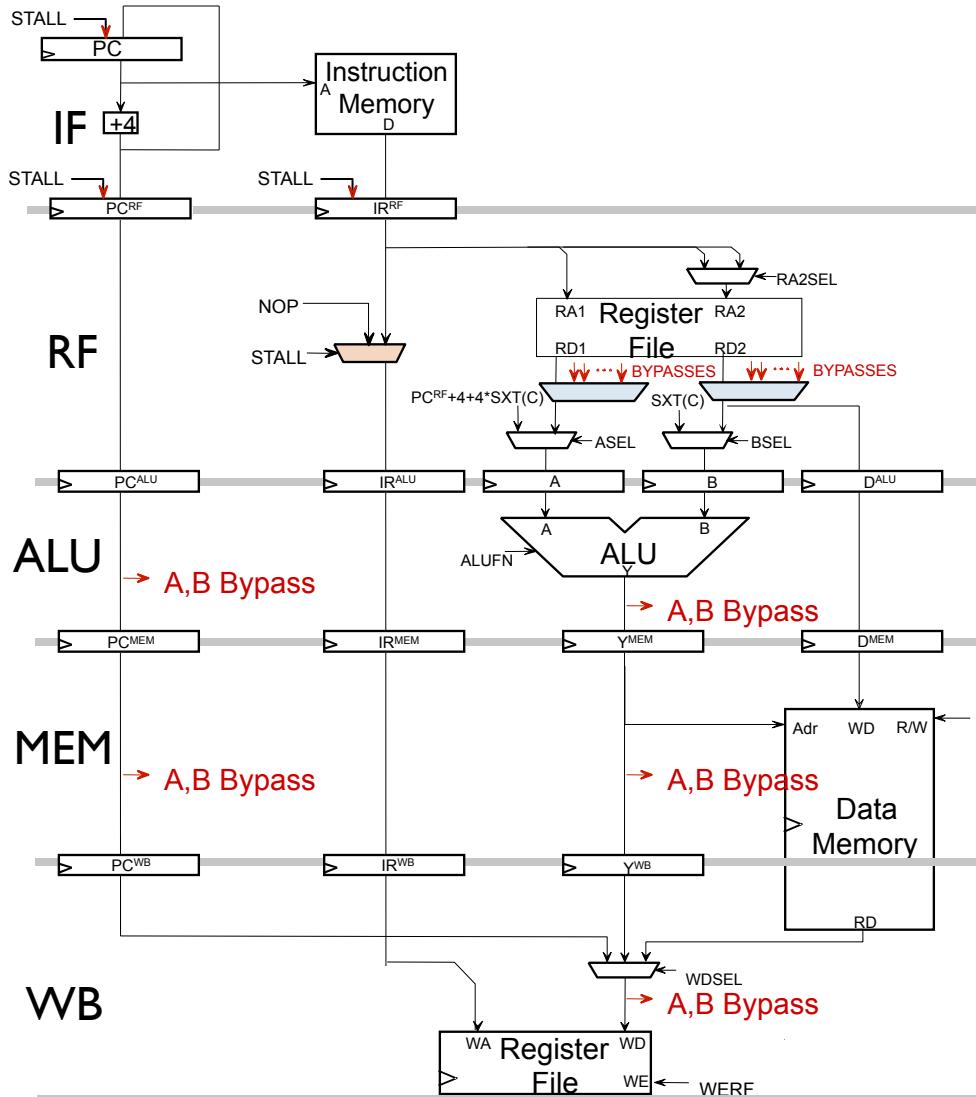
ADDC result computed ↑ R2 updated ↗

Bypass Logic



- Add bypass muxes to RF outputs
- Route ALU, MEM, WB outputs to mux inputs
- Bypass value if destination register of instruction in ALU, MEM, or WB matches source register of instruction in RF
 - But not R31!?
- What to do if multiple matches?
 - Select value from most recent instruction! (ALU > MEM > WB)

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?

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:

	I	2	3	4	5	6	7
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

Stall

LD data available ↑ R2 updated ↗

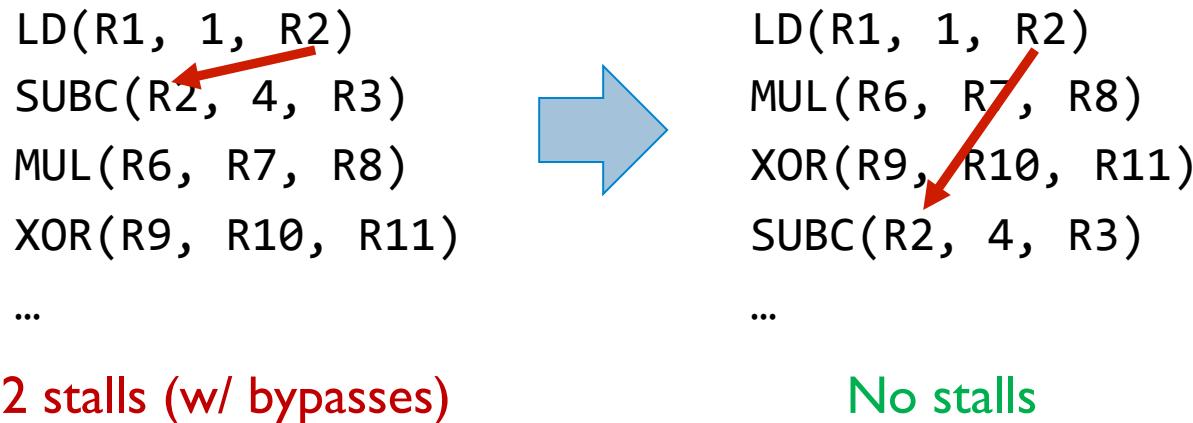
LD(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
...

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

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!

Or take the lazy route...

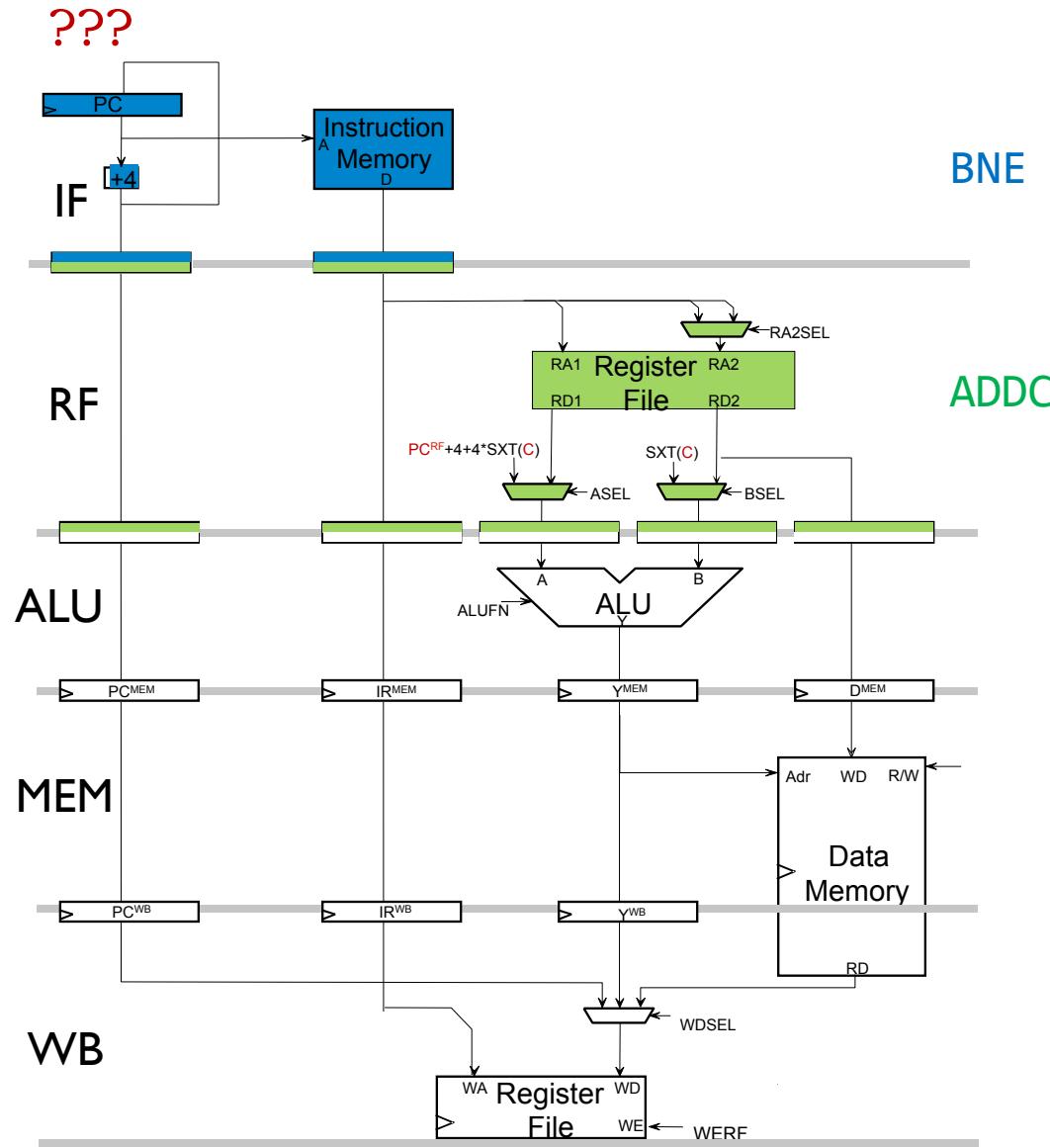
- Don't stall or bypass, just change the ISA so that registers are updated with a 3-instruction delay!
 - Compiler writers will love this!
 - Programmers will love this!
 - You will love this when you decide to release an 8-stage pipelined processor!



Roger Schultz
(CC-BY 2.0)

- ISAs outlive implementations, this is a bad idea

Control Hazards



BNE loop: ADDC(R1, 4, R2)
 BNE(R3, loop)
 SUB(R6, R7, R8)

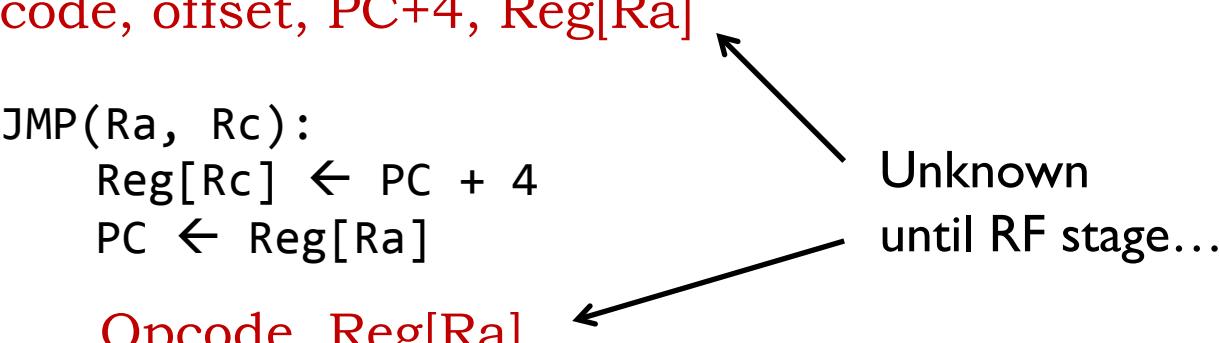
...

How do we set NextPC?

Control Hazards

- What do we need to compute NextPC?
 - BEQ/BNE: BEQ(Ra, label, Rc):
$$\begin{aligned} \text{Reg[Rc]} &\leftarrow \text{PC} + 4 \\ \text{if } (\text{Reg[Ra]} == 0) \\ &\quad \text{PC} \leftarrow \text{PC} + 4 + 4 * \text{SXT(offset)} \\ \text{else} \\ &\quad \text{PC} \leftarrow \text{PC} + 4 \end{aligned}$$

Opcode, offset, PC+4, Reg[Ra]



Opcode, Reg[Ra]

 - JMP: JMP(Ra, Rc):
$$\begin{aligned} \text{Reg[Rc]} &\leftarrow \text{PC} + 4 \\ \text{PC} &\leftarrow \text{Reg[Ra]} \end{aligned}$$

Opcode, Reg[Ra]

Unknown until RF stage...
 - All other instructions: **Opcode, PC+4**
 - (Exceptions also change PC, we'll deal with them later)

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

Resolving Control Hazards With Stalls

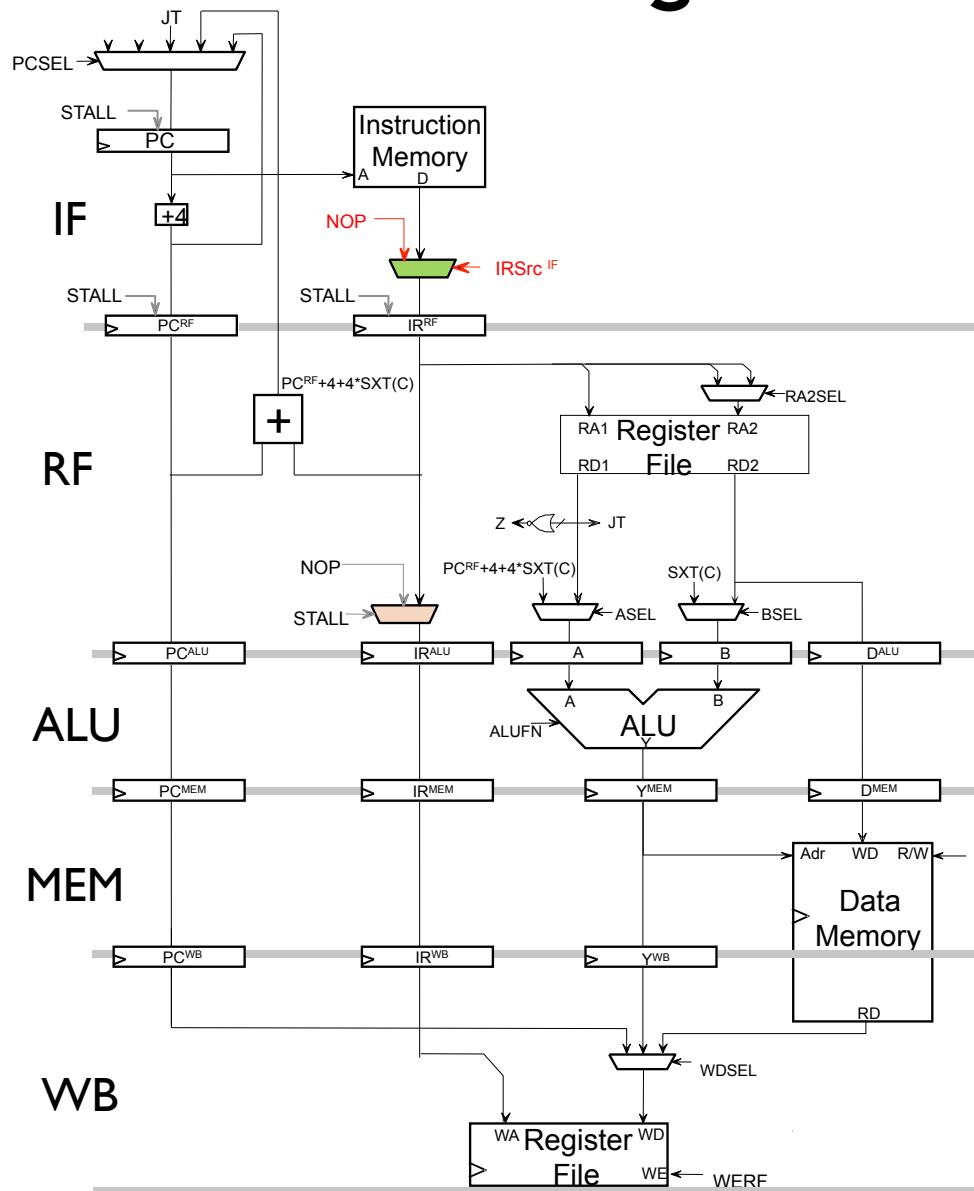
- If branch or jump in IF, stall IF for one cycle
 - Assume BNE is always taken in example code
- loop: ADDC(R1, -1, R3)
 MUL(R4, R5, R6)
 BNE(R3, loop)
 SUB(R6, R7, R8)
 ...

	I	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

↑ R3!=0 → Taken ↑ R3!=0

- Steady-state CPI?

Stall Logic For Control Hazards



- IRSrc^{IF} control signal
- If $\text{opcode}^{\text{RF}} == \text{JMP}, \text{BEQ}, \text{BNE}$
 - IRSrc^{IF}=1, inject NOP
 - Set PCSEL to load branch or jump target

ISA Issue: Simple vs Complex Branches

- Beta has very simple branch condition
 - $\text{Reg}[\text{Ra}] == 0$ 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?

	I	2	3	4	5	6	7	8
IF	ADDC	MUL	BNE	SUB	...	ADDC	MUL	BNE
RF		ADDC	MUL	BNE	SUB	NOP	ADDC	MUL
ALU			ADDC	MUL	BNE	NOP	NOP	ADDC
MEM				ADDC	MUL	BNE	NOP	NOP
WB					ADDC	NOP	MUL	BNE

More annulments (but sometimes fewer instructions)

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

Resolving Hazards with Speculation

- What's a good guess for NextPC?

PC+4

loop: ADDC(R1, -1, R3)
MUL(R4, R5, R6)
BNE(R3, loop)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
...

- Assume BNE is not taken in example

	I	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

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)
...

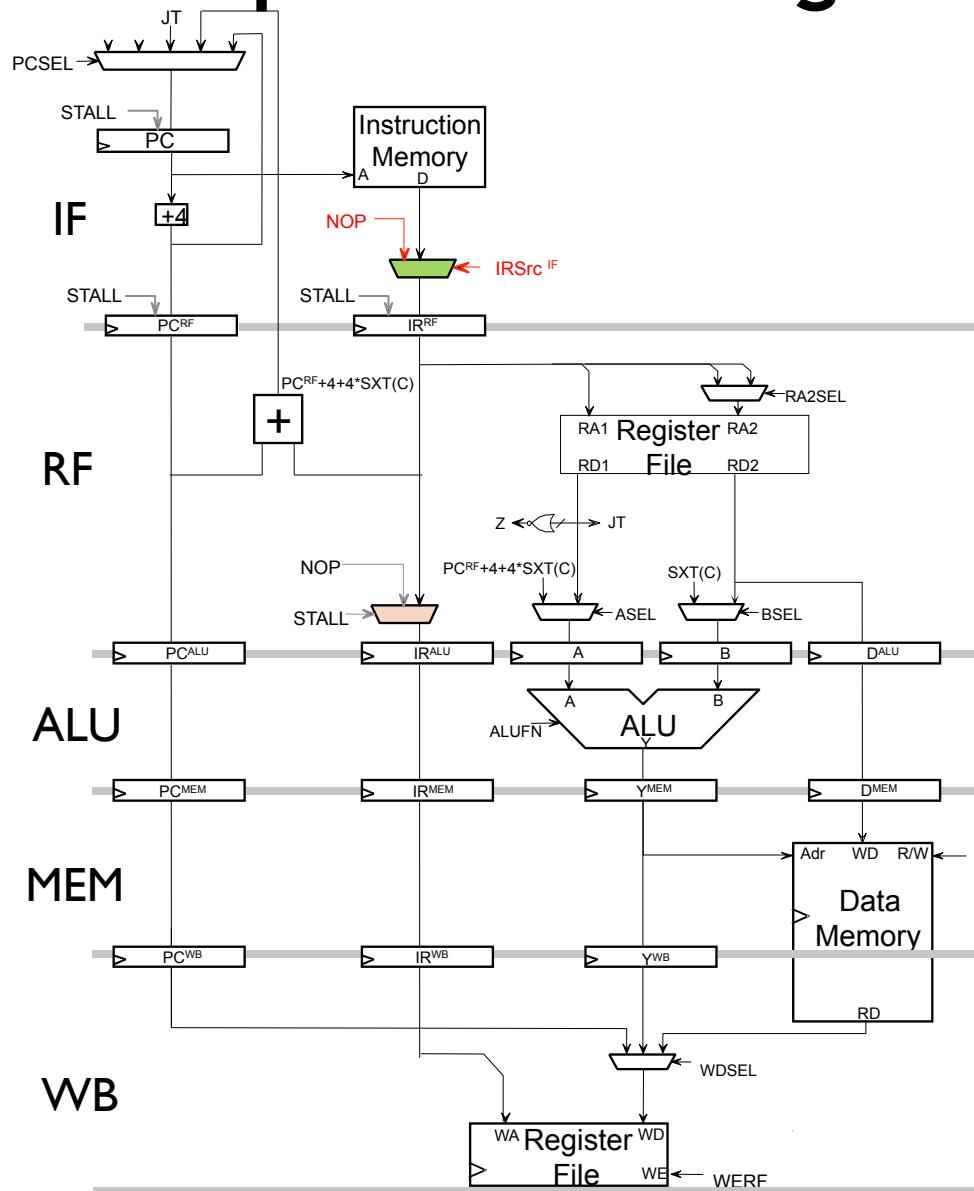
	I	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	SUB	ADDC	MUL	BNE	SUB	ADDC
RF		ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP
ALU			ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM				ADDC	MUL	BNE	NOP	ADDC	MUL
WB					ADDC	MUL	BNE	NOP	ADDC

Start fetching at PC+4 (SUB) but
BNE not resolved yet...



Guessed wrong, annul SUB

Speculation Logic For Control Hazards



- This looks familiar...
- **IRSrc^{IF}** control signal
- If $\text{opcode}^{\text{RF}} == \text{JMP}$ or **taken BEQ/BNE**
 - $\text{IRSrc}^{\text{IF}} = 1$, inject NOP to annul fetched inst. (aka “branch annulment”)
 - Set **PCSEL** to load branch or jump target

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

Branch Delay Slots

- Change the ISA so that the instruction following a jump or branch is always executed

Delay slot instruction executes
regardless of branch outcome

loop: ADDC(R1, -1, R3)
BNE(R3, loop)
MUL(R4, R5, R6)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
...

	I	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

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

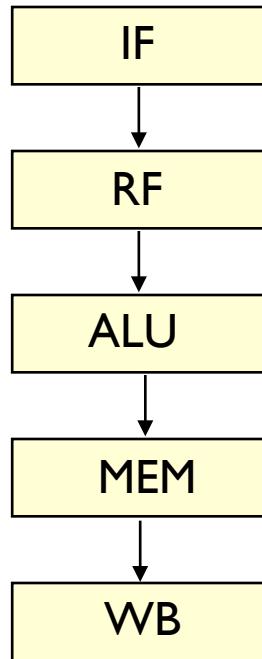


- ISAs outlive implementations, this is a bad idea

Exceptions

- On an exception, need to:
 - Save current PC+4 in XP (R30)
 - Load PC with exception vector (IllOp 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

When Can Exceptions Happen?



Memory fault (e.g., illegal memory address)

Illegal instruction

Arithmetic exception (e.g., divide by zero)

Memory fault (e.g., illegal memory address)

- Instructions following the one that causes the exception may already be in the pipeline...
- ... but none has written registers or memory yet ☺

Resolving Exceptions

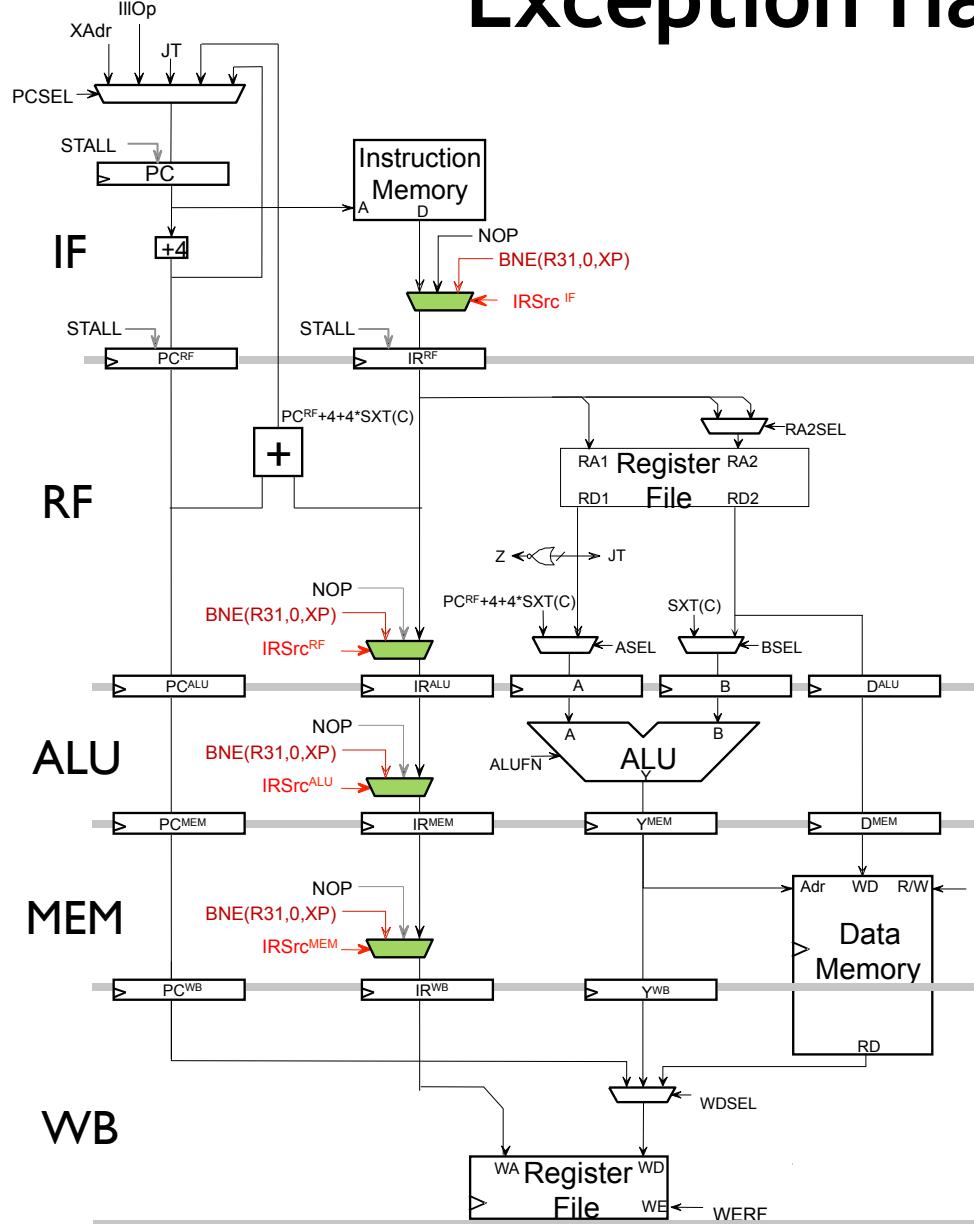
- If an instruction has an exception at stage i
 - Turn that instruction into BNE(R31, 0, XP) to save PC+4
 - Annul instructions in stages $i-1, \dots, 1$ (flush the pipeline)
 - Set PC \leftarrow IllOp or XAdr

- 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

Exception Handling Logic



- IRSrc^{IF,RF,ALU,MEM} muxes to inject NOP or BNE
 - NOP if preceding instruction has an exception
 - BNE if instruction in current stage has an exception

Multiple Exceptions?

Causes memory fault

→ LD(R1, 4, R2) Xaddr: ADDC IllOp: XORC
R1 ← R2 + 4

Invalid opcode → ...
MUL (R4, R5, R6) ...

、 、 、 、 、

	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!

Asynchronous Interrupts

Interrupts are easier:

// Interrupted code:

```
...  
LD(...)  
ADD(...)  
SUB(...) ← Interrupt Taken HERE  
...  
...
```

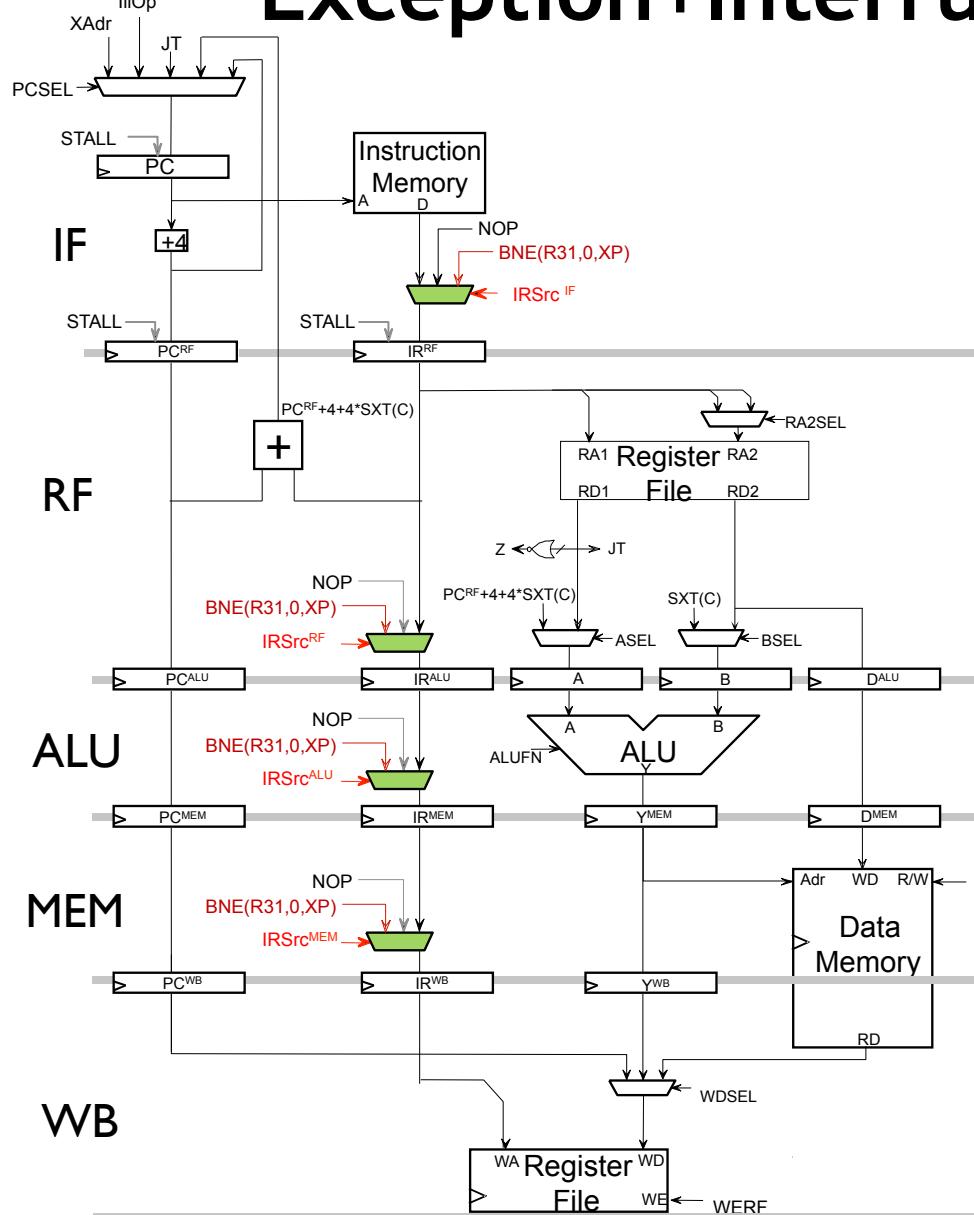
// Interrupt handler:

```
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

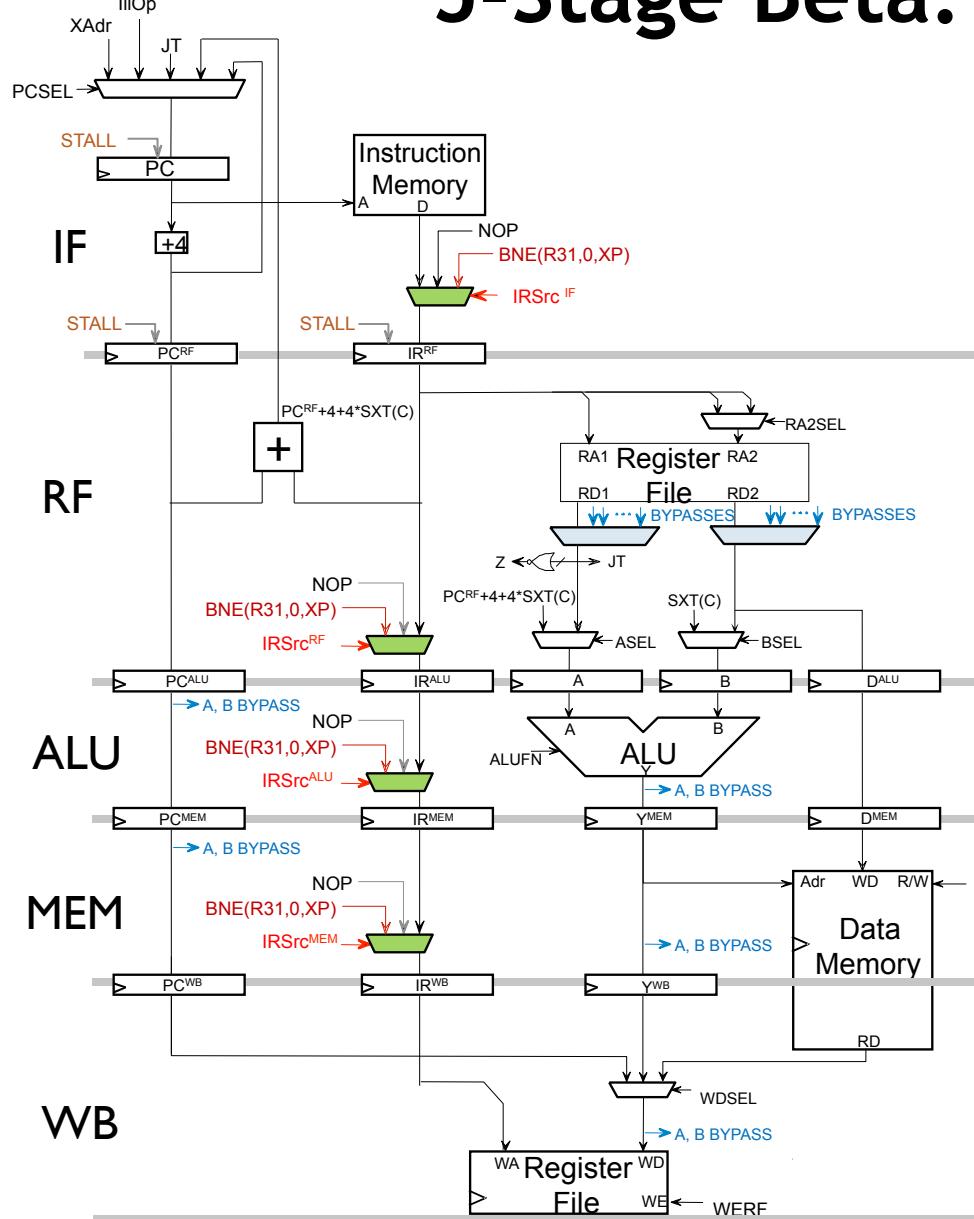
- Suppose interrupt is requested while SUB is in the IF stage (cycle 2)
- To handle:
 - Replace SUB instruction with BNE(...,XP)
 - Select Xadr as next PC
 - Code handler to return to SUB instruction
 - ADD and earlier insts. are unaffected

Exception+Interrupt Handling Logic



- Same as before
- IRSrc^{IF,RF,ALU,MEM} muxes to inject NOP or BNE
 - NOP if preceding instruction has an exception
 - BNE if instruction in current stage has an exception
- Use IRSrc^{IF} mux to inject BNE on an interrupt (same as an exception in IF)

5-Stage Beta: Final Version



- Data hazards:
 - Stall IF and RF ($\text{STALL}=1 + \text{IRSsrc}^{\text{RF}}=\text{NOP}$)
 - Bypass
- Control hazards: Speculate $\text{PC}+4$ and
 - JMP or taken branch in RF,
 - $\text{IRSsrc}^{\text{RF}}=\text{NOP}$
 - $\text{PCSEL} \rightarrow \text{JT}/\text{branch target}$
 - If exception at stage X
 - $\text{IRSsrc}^X=\text{BNE}$
 - Previous $\text{IRSsrc}^X=\text{NOP}$
 - $\text{PCSEL} \rightarrow \text{XAdr}$ or IIOp
 - If interrupt
 - $\text{IRSsrc}^{\text{IF}}=\text{BNE}$
 - $\text{PCSEL} \rightarrow \text{XAdr}$

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

16. Virtual Memory

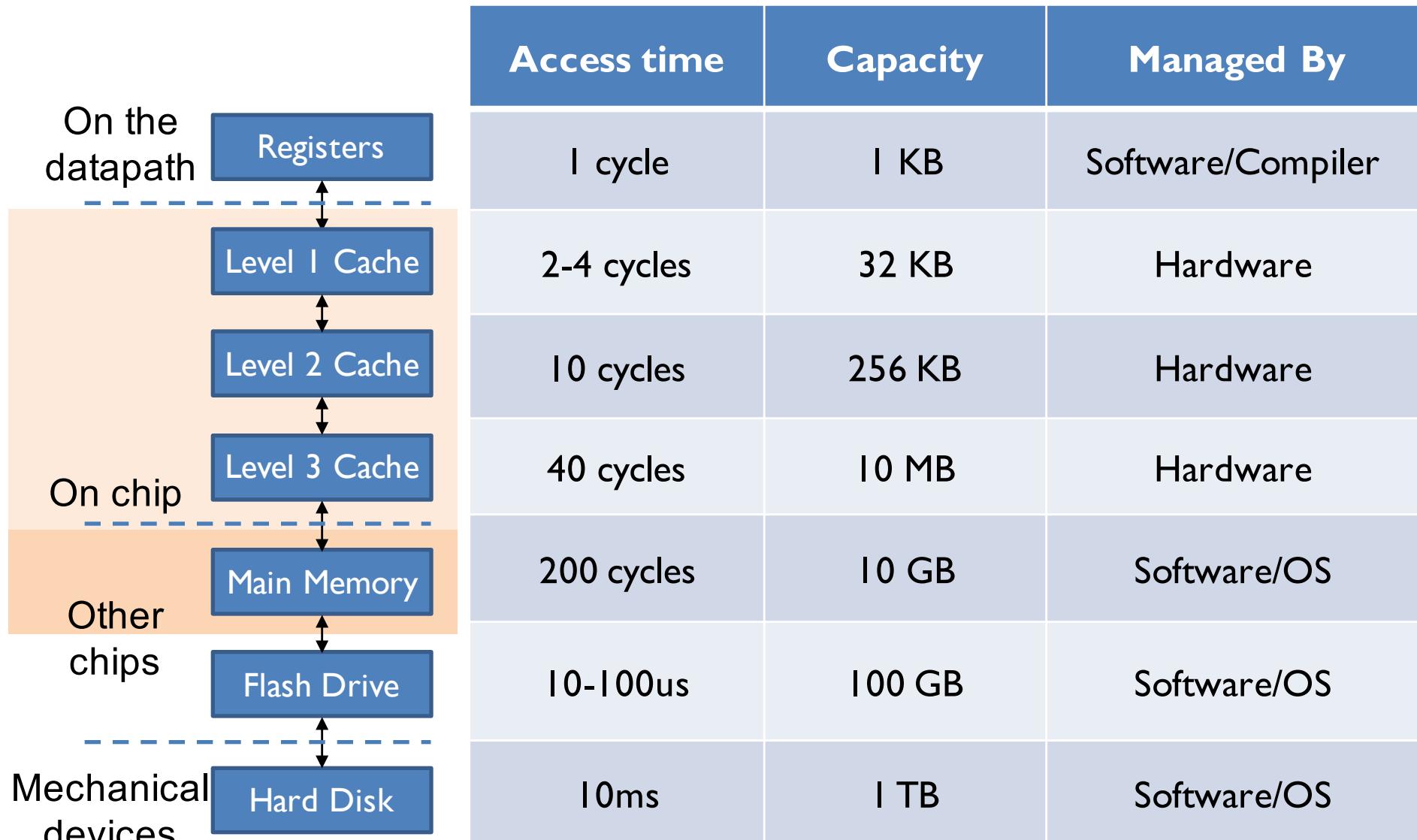
6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

Even more Memory Hierarchy

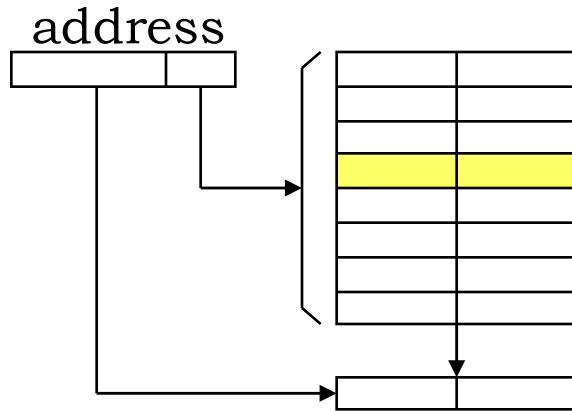
Reminder: A Typical Memory Hierarchy

- Everything is a cache for something else



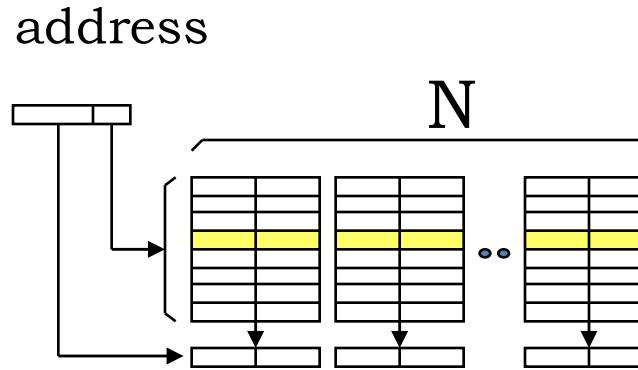
Reminder: Hardware Caches

Direct-mapped



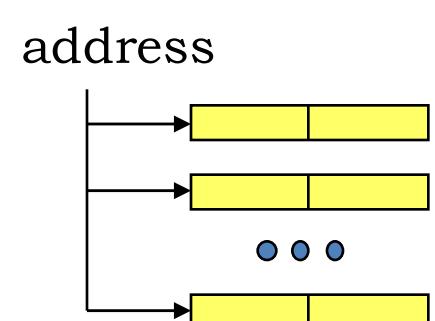
Each address maps to a single location in the cache, given by index bits

N-way set-associative



Each address can map to any of N locations (ways) of a single set, given by its index bits

Fully associative



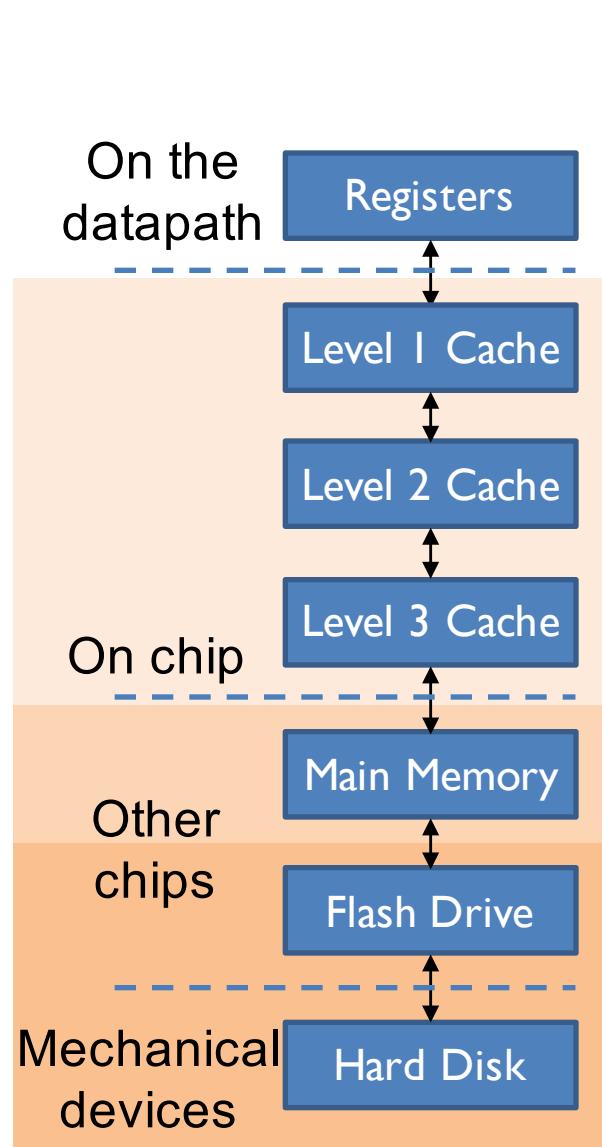
Any location can map to any address

Other implementation choices:

- Block size
- Replacement policy (LRU, Random, ...)
- Write policy (write-through, write-behind, write-back)

Reminder: A Typical Memory Hierarchy

- Everything is a cache for something else

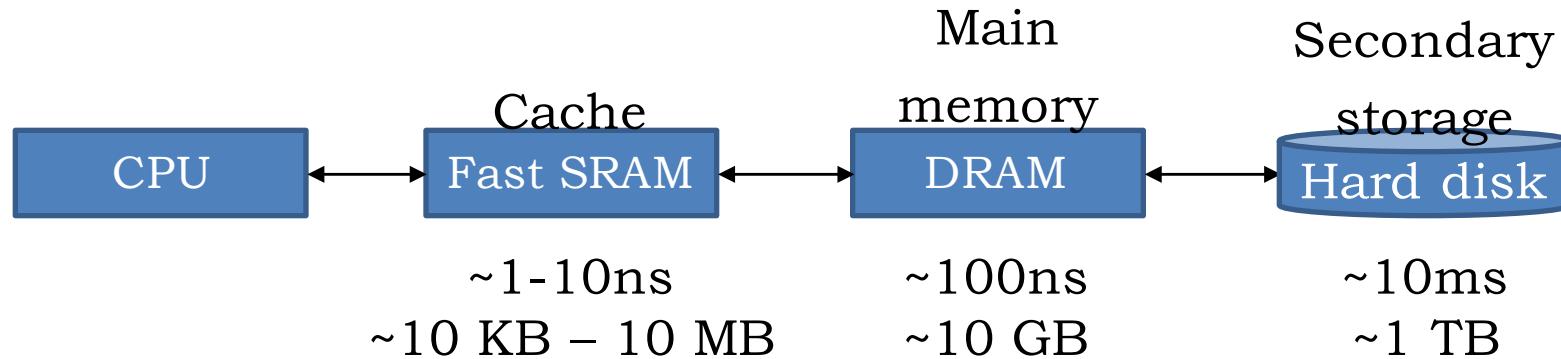


Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
~100 ns	16 KB	Hardware
~100 ns	1 MB	Hardware
~100 ns	16 MB	Hardware
~100 ns	1 GB	Software/OS
~10 ms	1 TB	Software/OS
~100 ms	1 TB	Software/OS

Before: Hardware Caches

TODAY: Virtual Memory

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**

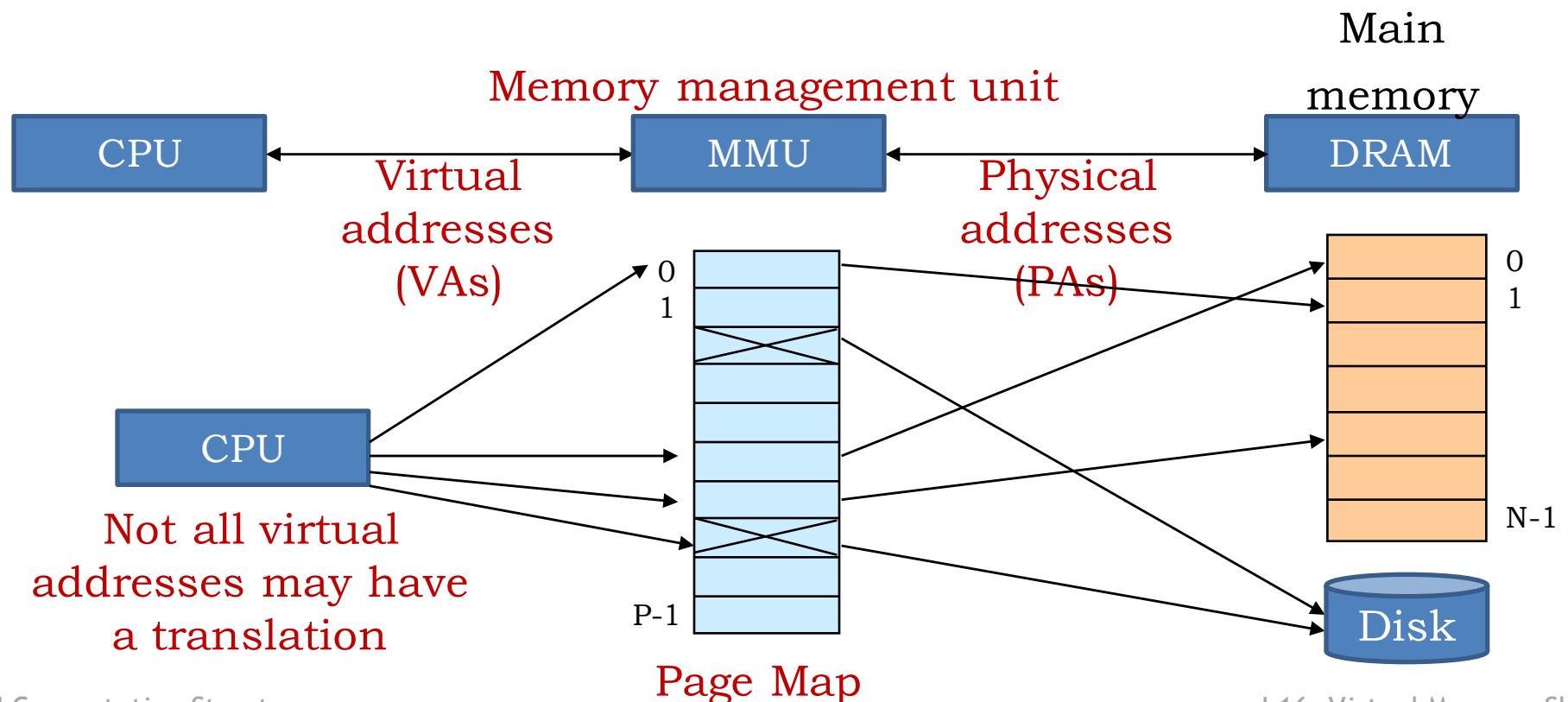
Impact of Enormous Miss Penalty

- 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 ($\sim 1\mu s$) vs 10ms?
 - Approach: Handle hits in hardware, misses in software
 - Simpler implementation, more flexibility

Basics of Virtual Memory

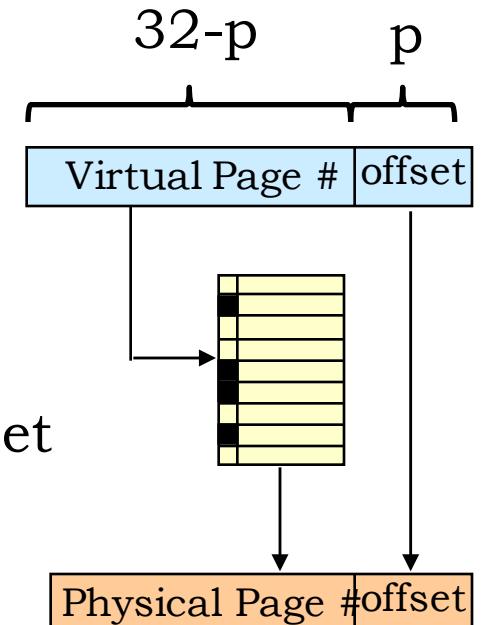
Virtual Memory

- 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**



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.



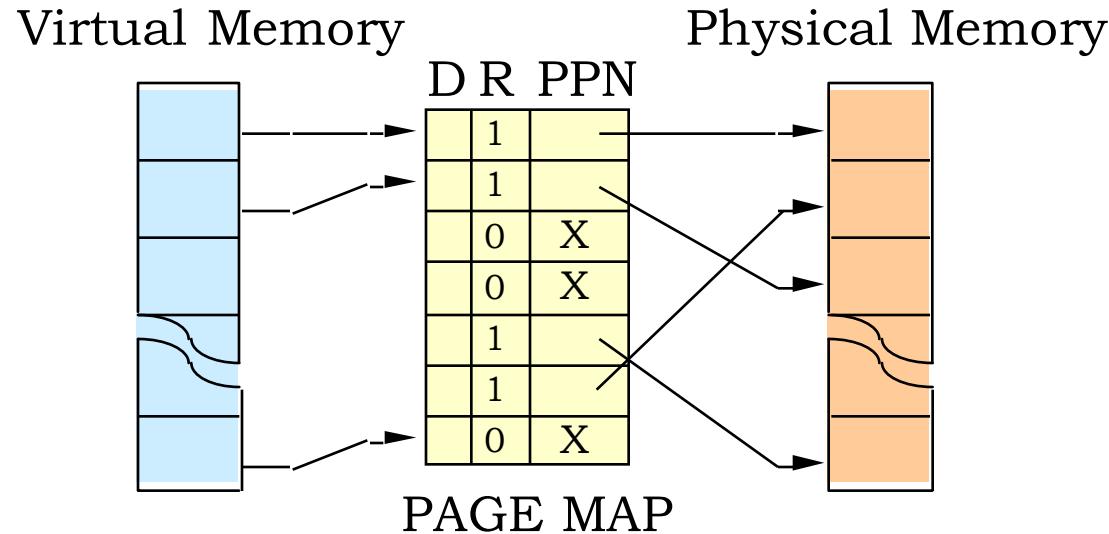
Using main memory as a page cache = *paging* or *demand paging*

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...

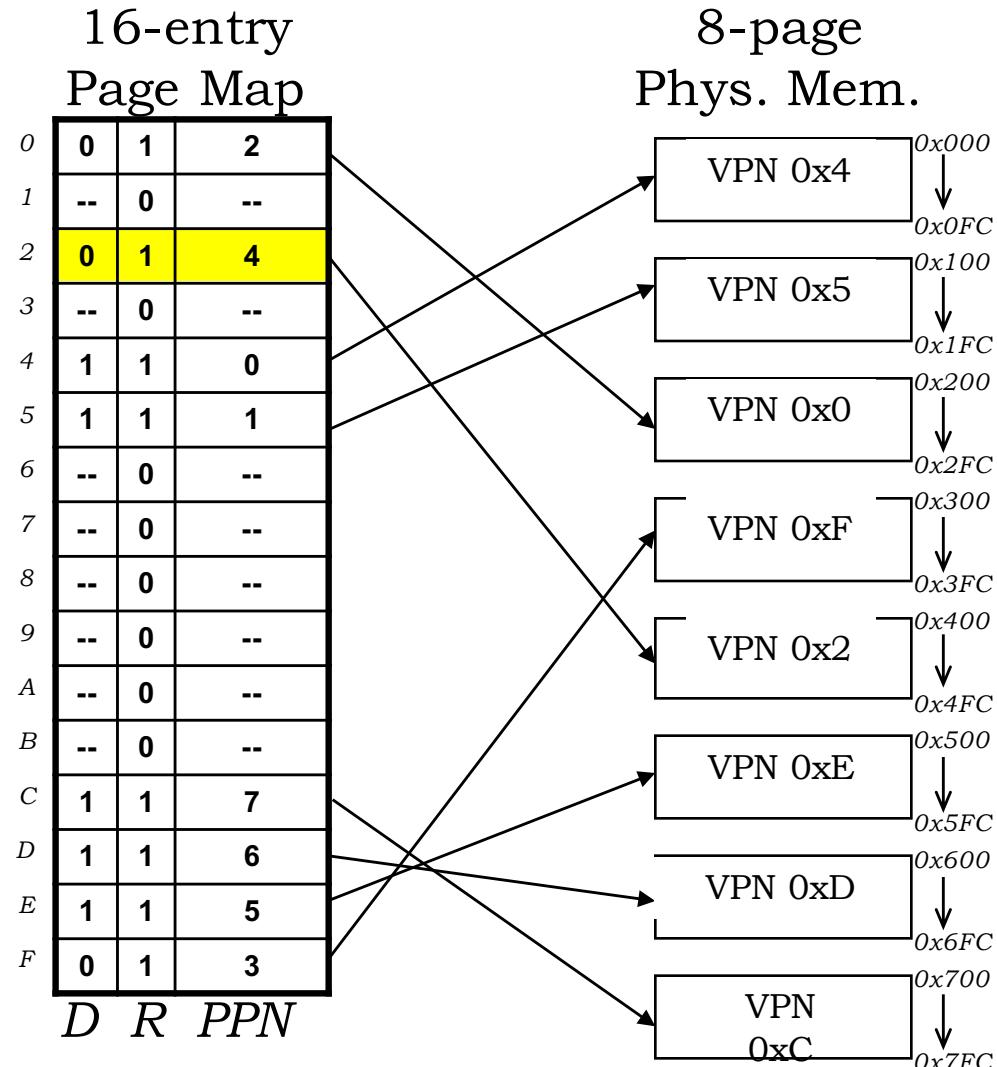
Simple Page Map Design



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)

Example: Virtual → Physical Translation



VPN	offset
4	8
3	8

VA
PA
PPN

Setup:

- 256 bytes/page (2^8)
- 16 virtual pages (2^4)
- 8 physical pages (2^3)
- 12-bit VA (4 vpn, 8 offset)
- 11-bit PA (3 ppn, 8 offset)
- LRU page: VPN = 0xE

LD(R31,0x2C8,R0):

VA = 0x2C8, PA = 0x4C8

VPN = 0x2
→ PPN = 0x4

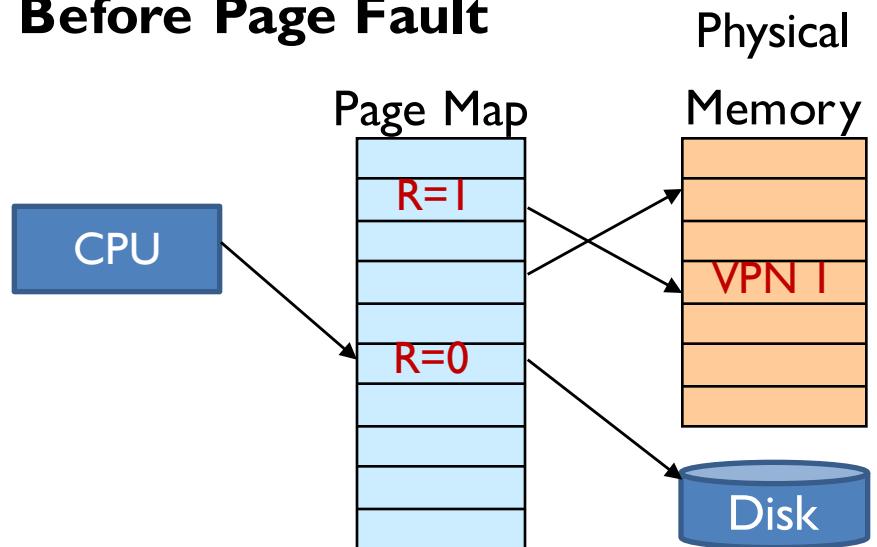
Page Faults

Page Faults

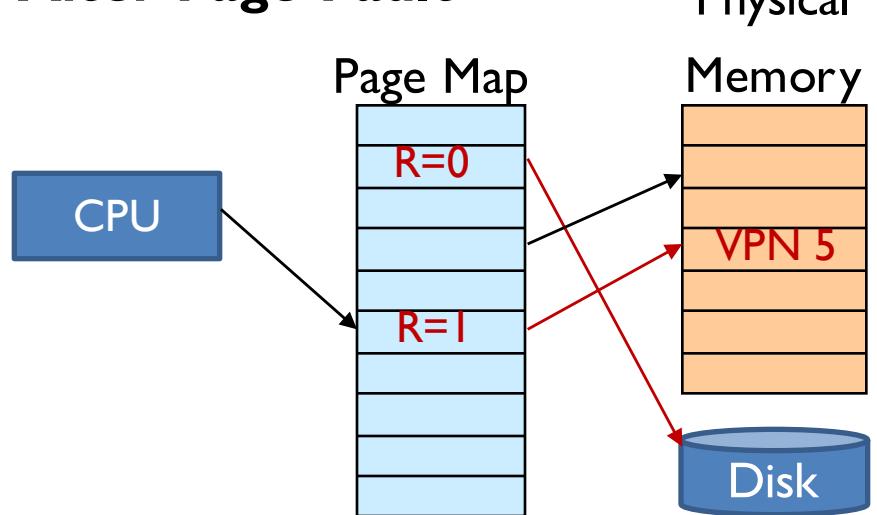
If a page does not have a valid translation, MMU causes a **page fault**. OS page fault handler is invoked, handles miss:

- 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

Before Page Fault

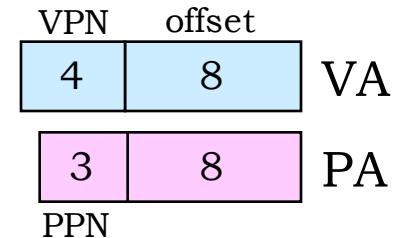
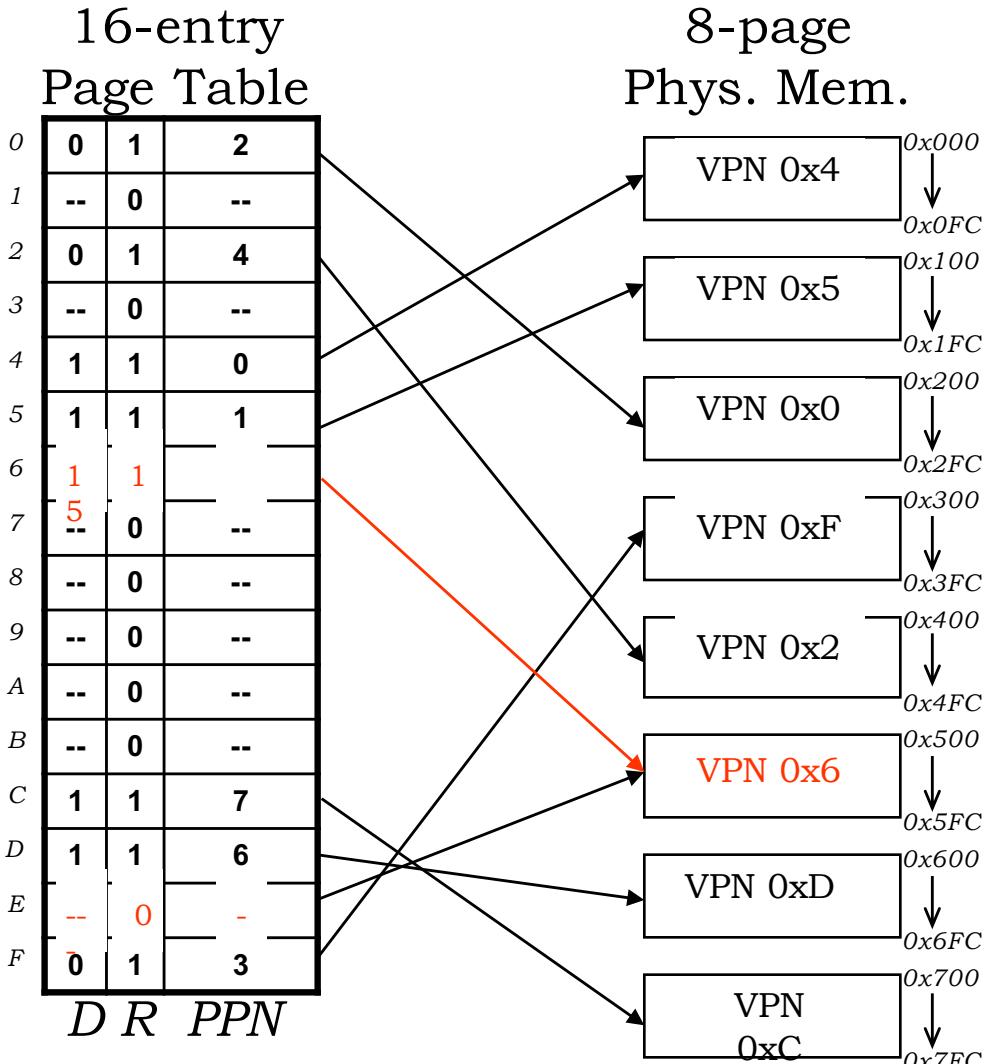


After Page Fault



** https://en.wikipedia.org/wiki/Page_replacement_algorithm#Page_replacement_algorithms

Example: Page Fault



Setup:

- 256 bytes/page (2^8)
- 16 virtual pages (2^4)
- 8 physical pages (2^3)
- 12-bit VA (4 vpn, 8 offset)
- 11-bit PA (3 ppn, 8 offset)
- LRU page: VPN = 0xE

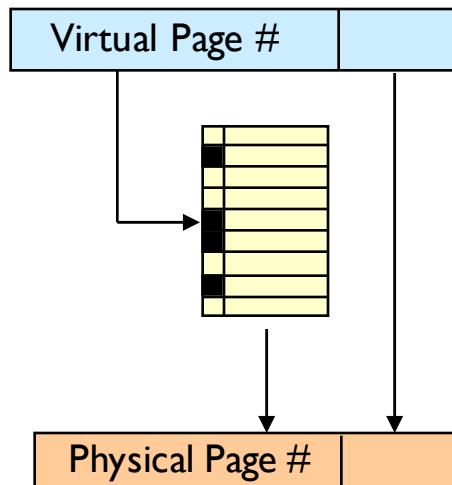
ST(BP,-4,SP), SP = 0x604
 VA = 0x600, PA = 0x500

VPN = 0x6

- ⇒ Not resident, it's on disk
- ⇒ Choose page to replace (LRU = 0xE)
- ⇒ D[0xE] = 1, so write 0x500-0x5FC to disk
- ⇒ Mark VPN 0xE as no longer resident
- ⇒ Read in page VPN 0x6 from disk into 0x500-0x5FC
- ⇒ Set up page map for VPN 0x6 = PPN 0x5
- ⇒ PA = 0x500
- ⇒ This is a write so set D[0x6] = 1

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 2^p , 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;  
}
```

The HW/SW Balance

IDEA:

- devote HARDWARE to high-traffic, performance-critical path
- use (slow, cheap) SOFTWARE to handle exceptional cases

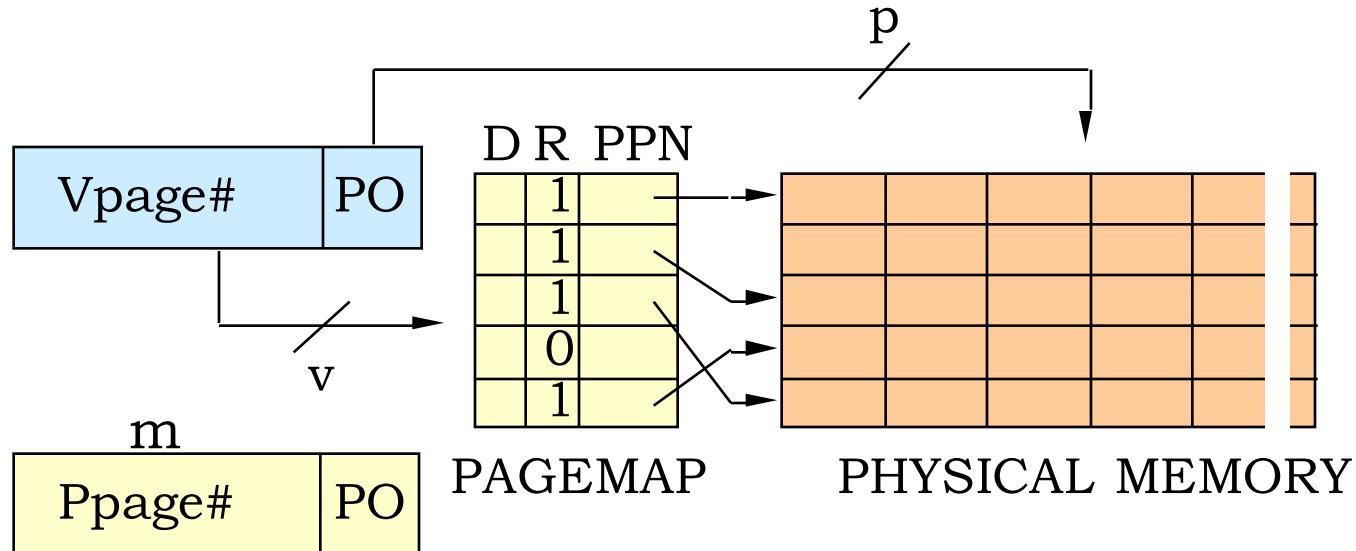
```
hardware {  
    int VtoP(int VPageNo,int PO) {  
        if (R[VPageNo] == 0)PageFault(VPageNo);  
        return (PPN[VPageNo] << p) | PO;  
    }  
  
    software {  
        /* Handle a missing page... */  
        void PageFault(int VPageNo) {  
            int i = SelectLRUPage();  
            if (D[i] == 1) WritePage(DiskAdr[i],PPN[i]);  
            R[i] = 0;  
  
            PA[VPageNo] = PPN[i];  
            ReadPage(DiskAdr[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

Building the MMU

Page Map Arithmetic



$(v + p)$ bits in virtual address

$(m + p)$ bits in physical address

2^v number of VIRTUAL pages

2^m number of PHYSICAL pages

2^p bytes per physical page

2^{v+p} bytes in virtual memory

2^{m+p} bytes in physical memory

$(m+2)2^v$ bits in the page map

Typical page size: 4KB -16 KB

Typical $(v+p)$: 32-64 bits

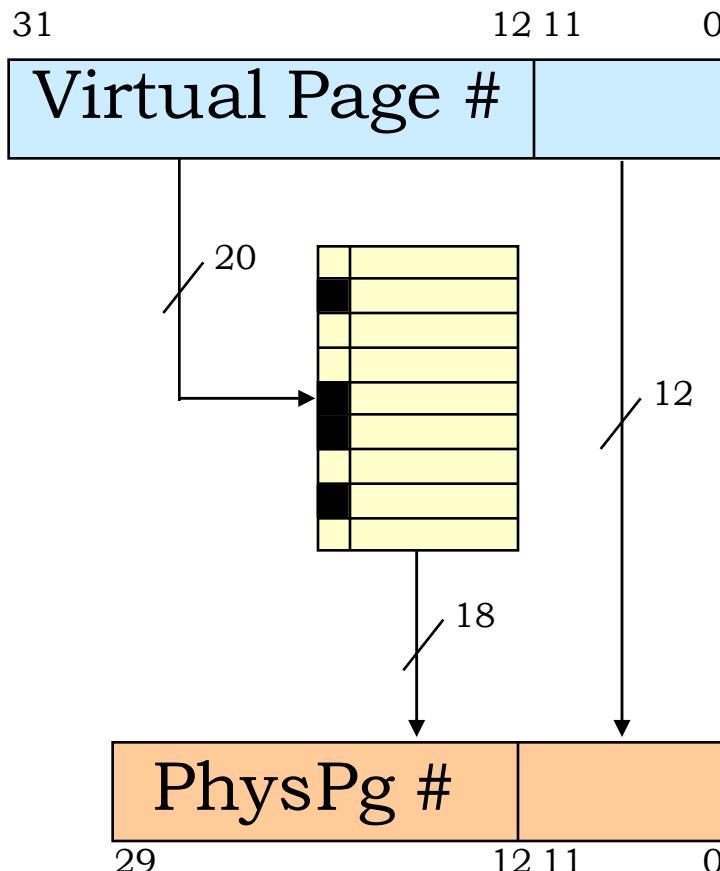
(4GB-16**EB**)

Typical $(m+p)$: 30-40+ bits

(1GB-1TB)

Long virtual addresses allow
ISAs to support larger
memories → ISA longevity

Example: Page Map Arithmetic



SUPPOSE...

32-bit Virtual address ($v+p$)

30-bit physical address ($m+p$)

4 KB page size ($p = 12$)

THEN:

$$\# \text{ Physical Pages} = \underline{2^{18} = 256\text{K}}$$

$$\# \text{ Virtual Pages} = \underline{2^{20}}$$

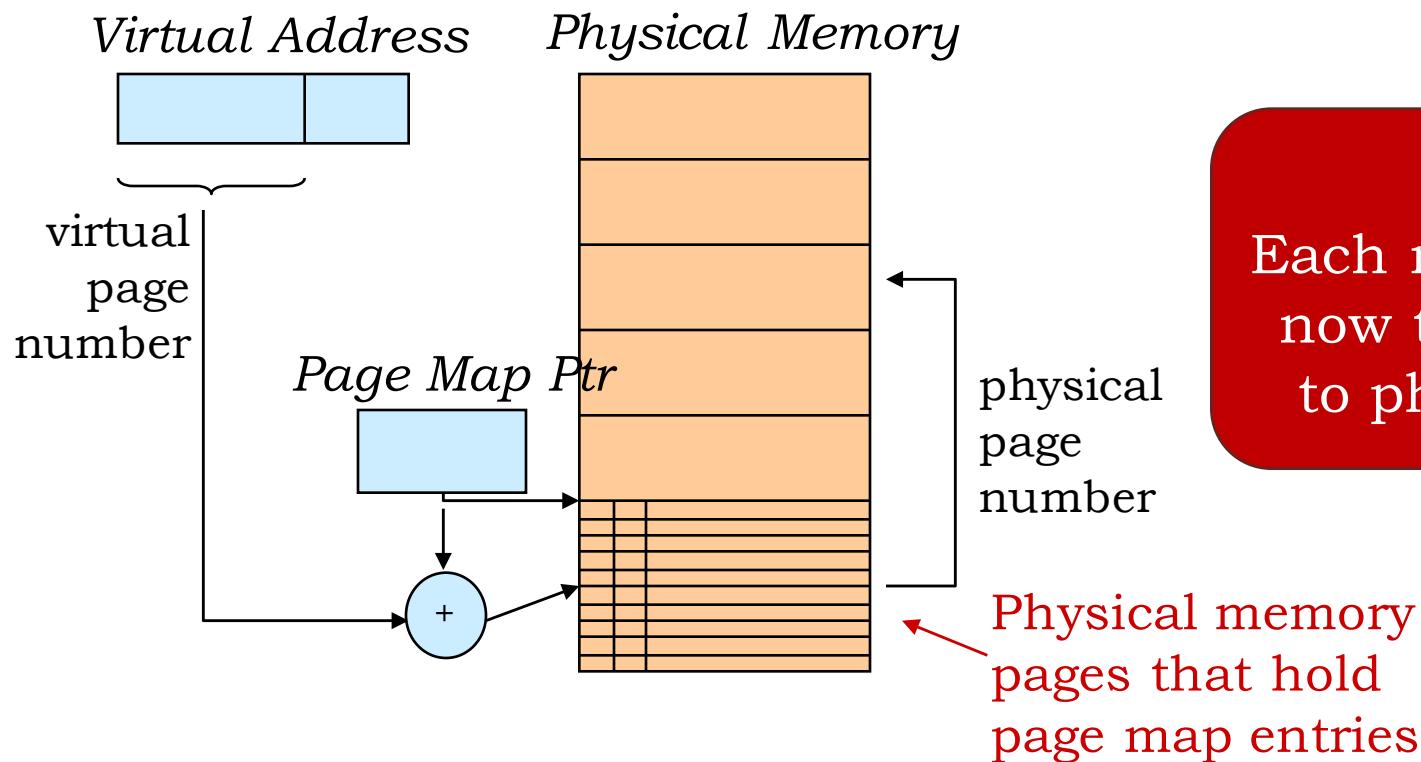
$$\# \text{ Page Map Entries} = \underline{2^{20} = 1\text{M}}$$

$$\# \text{ Bits In pagemap} = \underline{20 * 2^{20} \sim 20\text{M}}$$

Use fast SRAM for page map??? OUCH!

RAM-Resident Page Maps

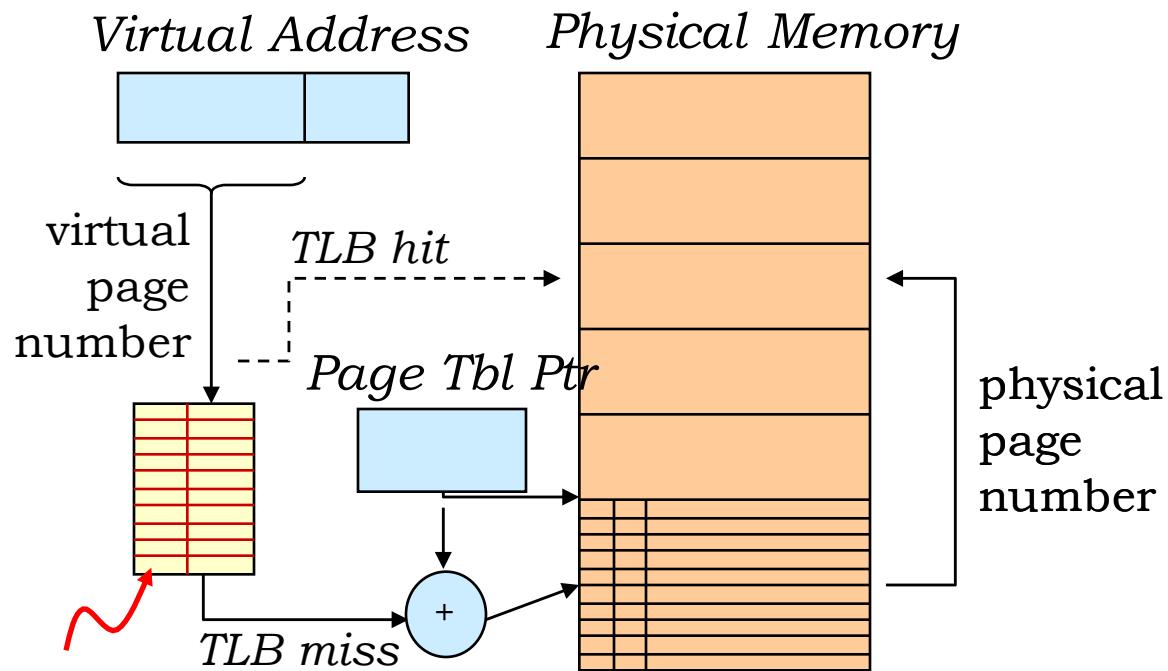
- Small page maps can use dedicated SRAM... gets expensive for big ones!
- Solution: Move page map to main memory:



PROBLEM
Each memory reference
now takes 2 accesses
to physical memory!

Translation Look-aside Buffer (TLB)

- Problem: 2x performance hit... each memory reference now takes 2 accesses!
- Solution: Cache the page map entries



TLB: small cache of page table entries
Associative lookup by VPN

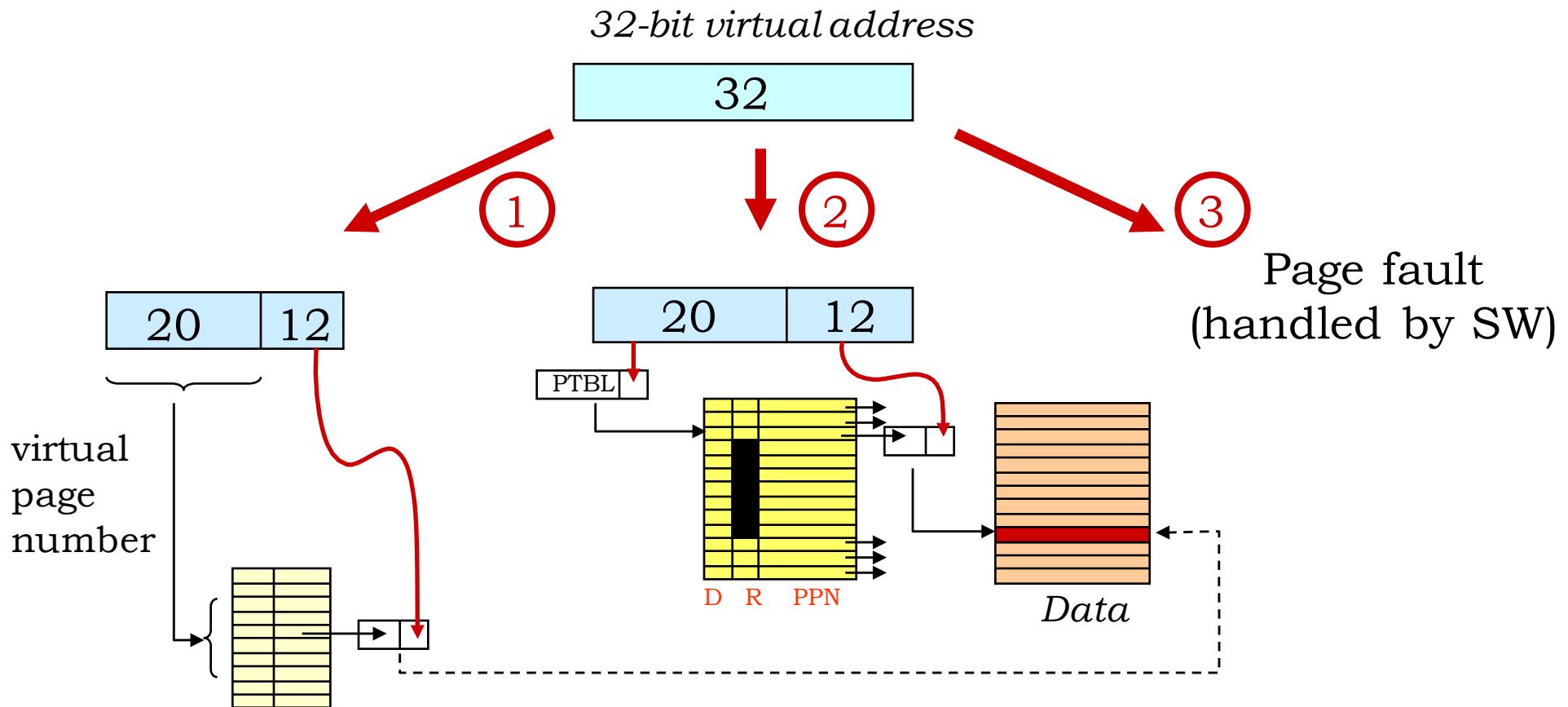
IDEA:
LOCALITY in memory reference patterns → SUPER locality in references to page map

VARIATIONS:

- multi-level page map
- paging the page map!

https://en.wikipedia.org/wiki/Translation_lookaside_buffer

MMU Address Translation



Look in TLB: VPN \rightarrow PPN cache
Usually implemented as a small
fully-associative cache

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]

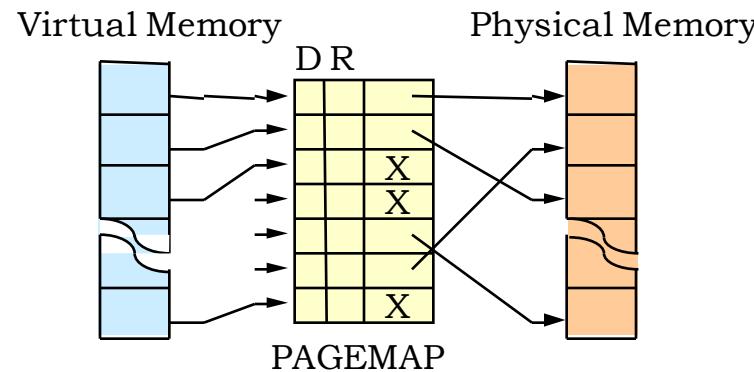
TLB		Page Map		
Tag	Data	VPN	R	D
VPN	R D PPN	PPN		
0	0 0 3	0	0 0 7	
6	1 1 2	1	1 1 9	
1	1 1 9	2	1 0 0	
3	0 0 5	3	0 0 5	
		4	1 0 5	
		5	0 0 3	
		6	1 1 2	
		7	1 0 4	
		8	1 0 1	
			...	

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

Contexts

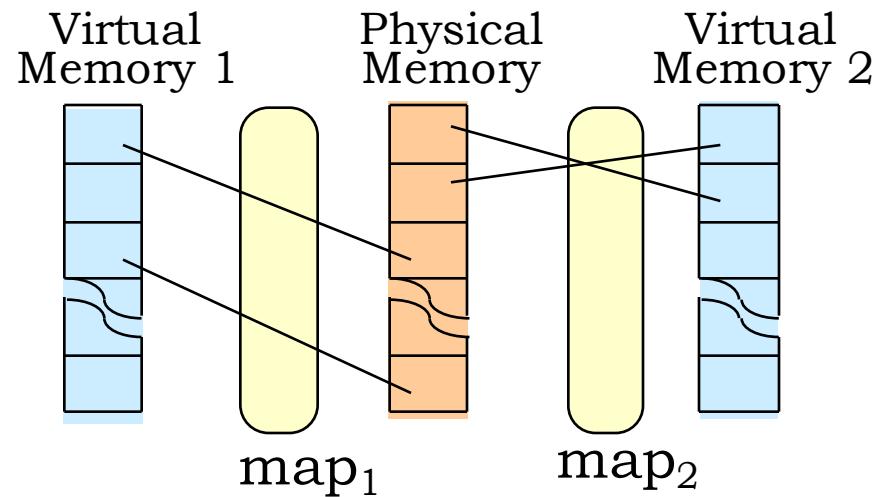
Contexts

A context is a mapping of VIRTUAL to PHYSICAL locations, as dictated by contents of the page map:

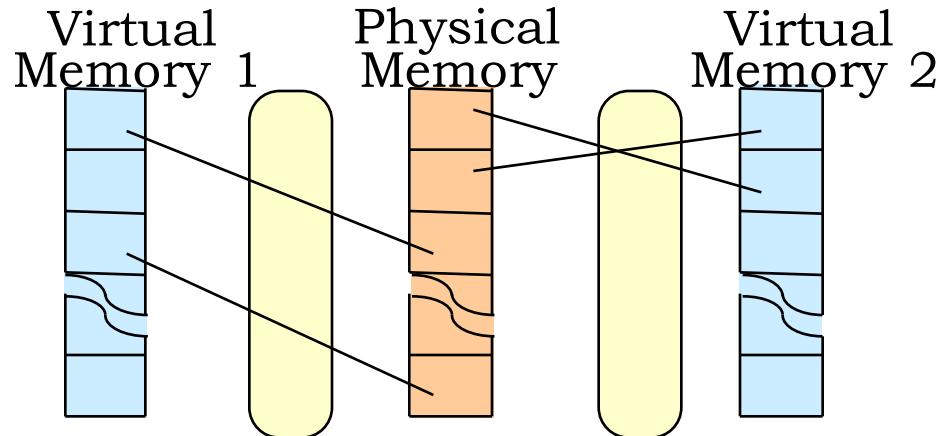


Several programs may be simultaneously loaded into main memory, each in its separate context:

“Context switch”:
reload the page map?



Contexts: A Sneak Preview

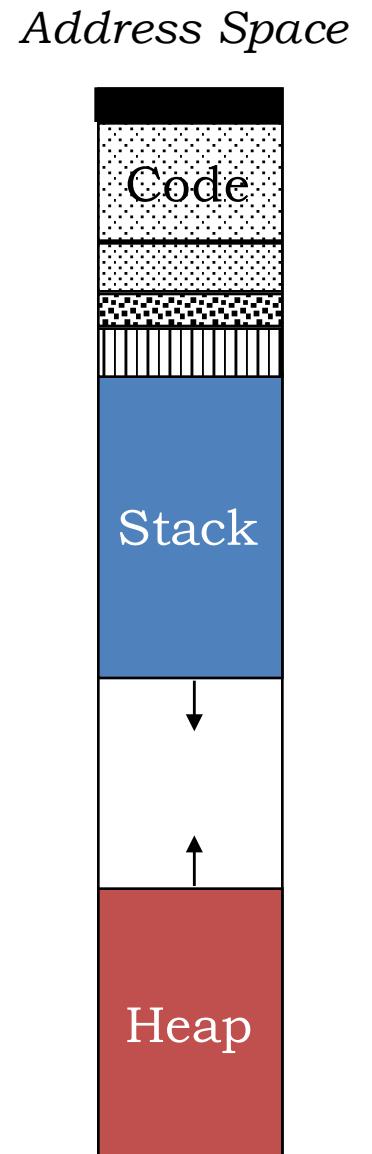


First Glimpse of a
VIRTUAL MACHINE

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.

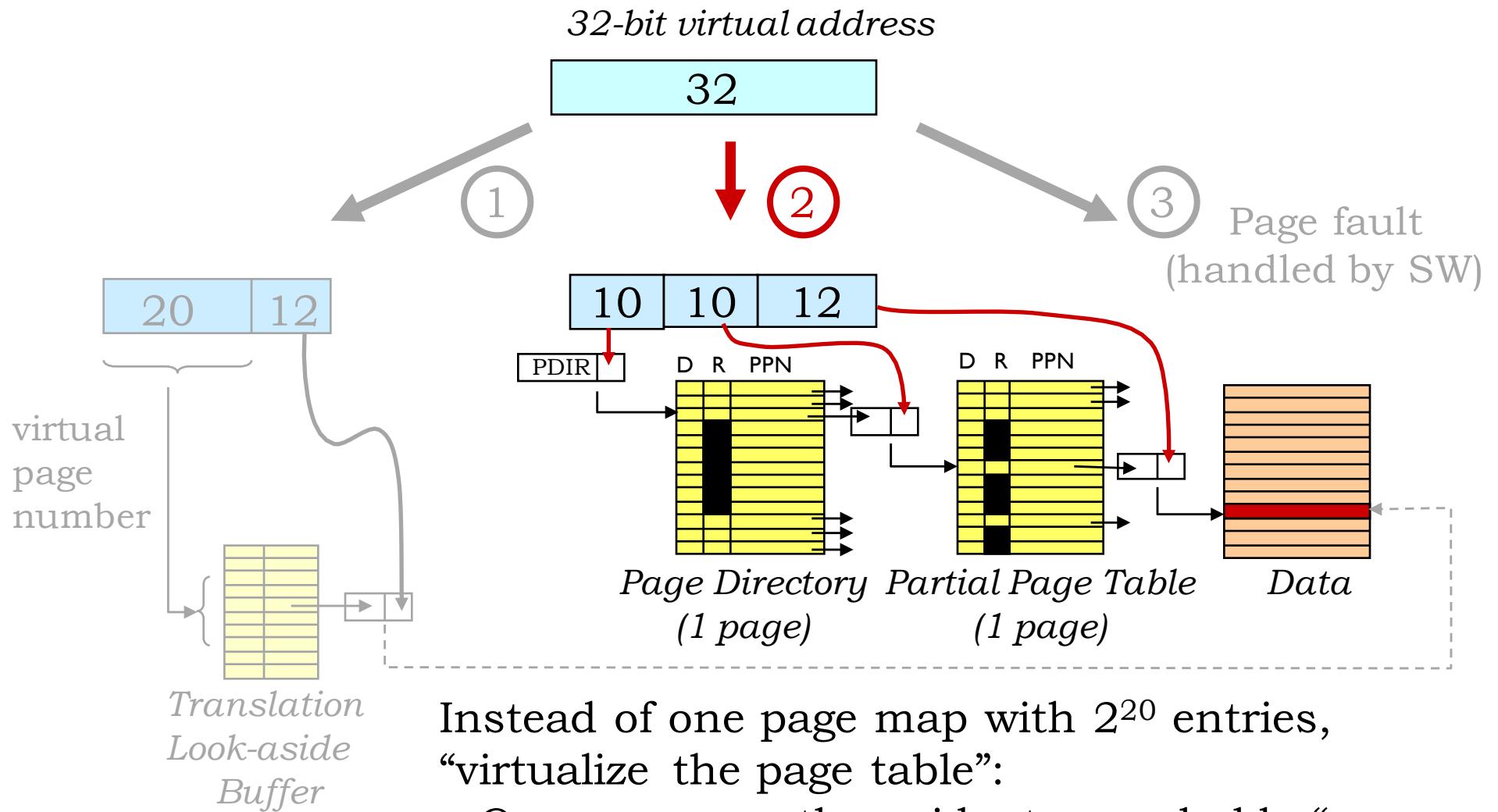
Memory Management & Protection

- 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



MMU Improvements

Multi-level Page Maps

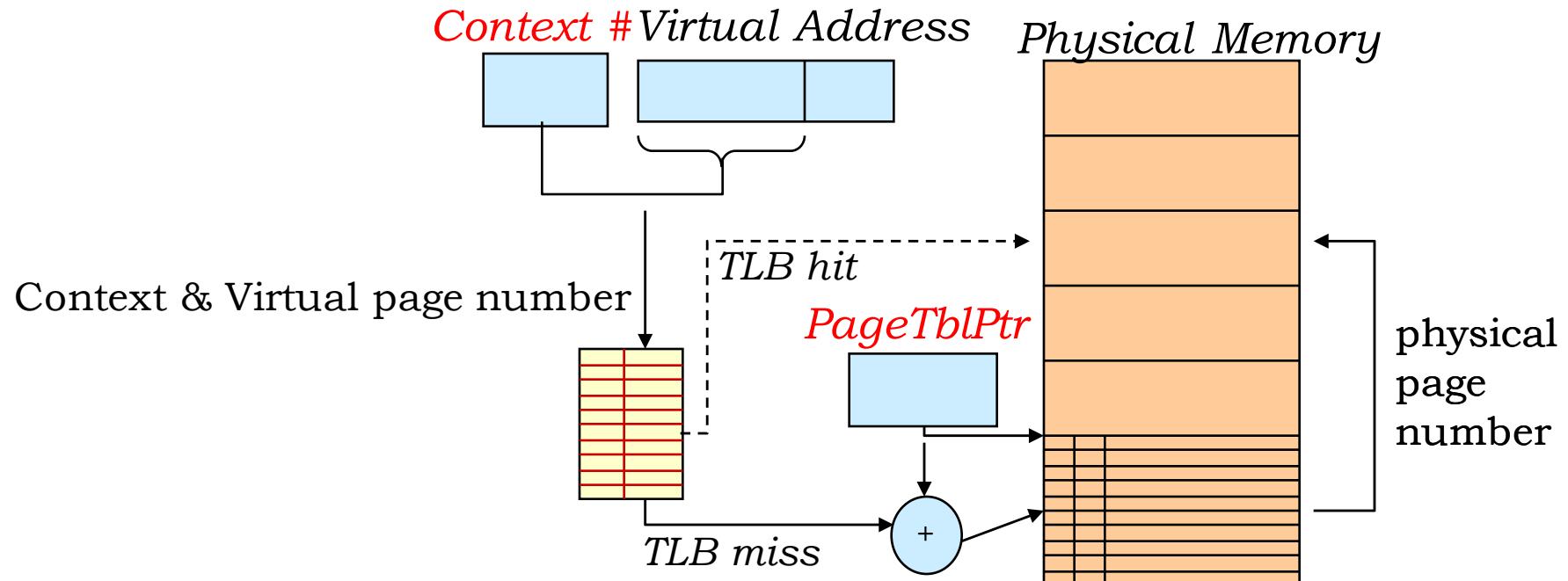


Instead of one page map with 2^{20} entries,
“virtualize the page table”:

One permanently-resident page holds “page directory” which has 1024 entries pointing to 1024-entry partial page tables in *virtual* memory!

Rapid Context-Switching

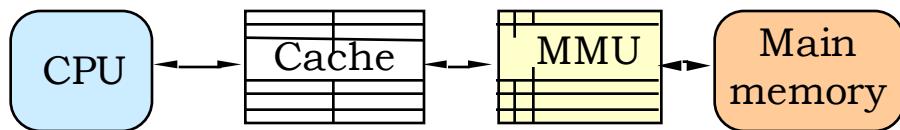
Add a register to hold index of current context. To switch contexts: update Context # and PageTblPtr registers. Don't have to flush TLB since each entry's tag includes context # in addition to virtual page number



Using Caches with Virtual Memory

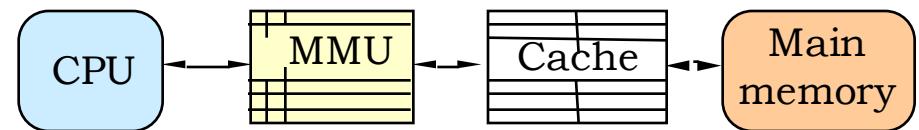
Virtually-Addressed Cache

Tags from virtual addresses



Physically-Addressed Cache

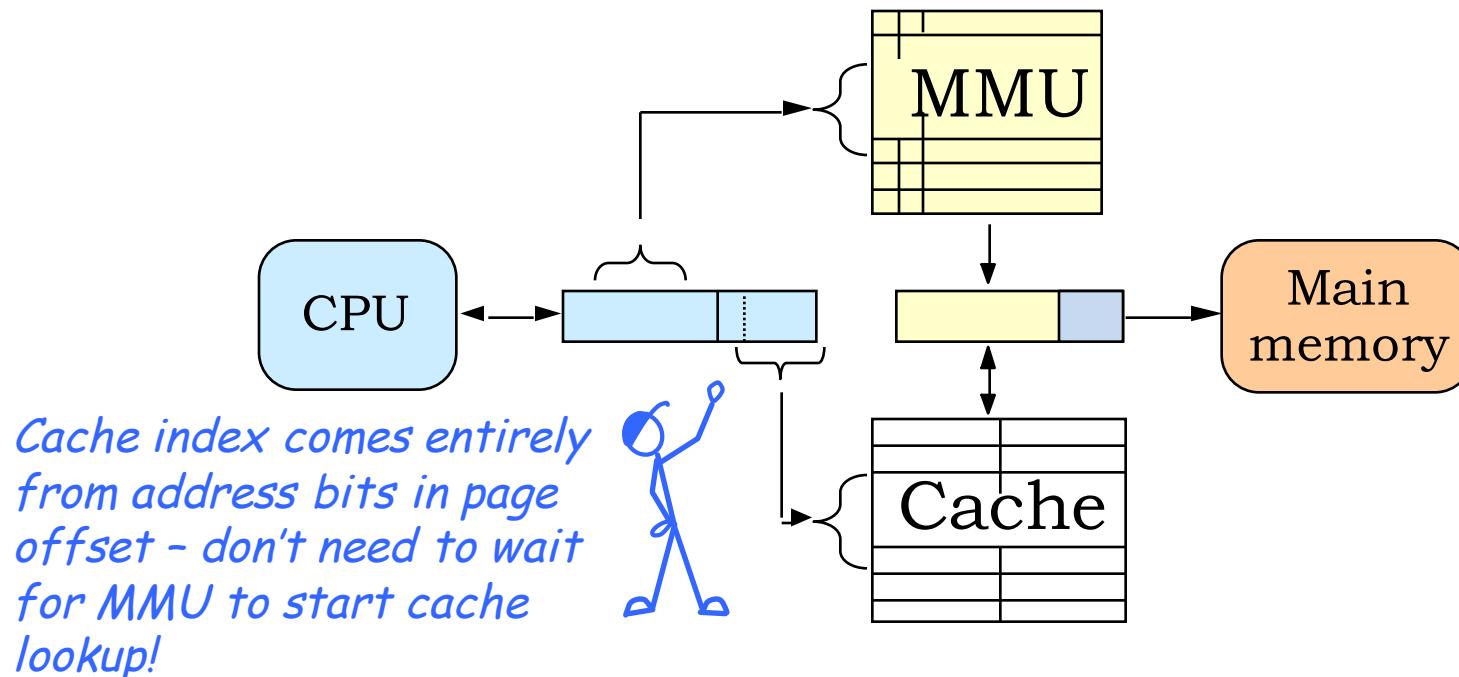
Tags from physical addresses



- FAST: No MMU time on HIT
- Problem: Must flush cache after context switch

- Avoids stale cache data after context switch
- SLOW: MMU time on HIT

Best of Both Worlds: Physically-Indexed, Virtually-Tagged Cache



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

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!

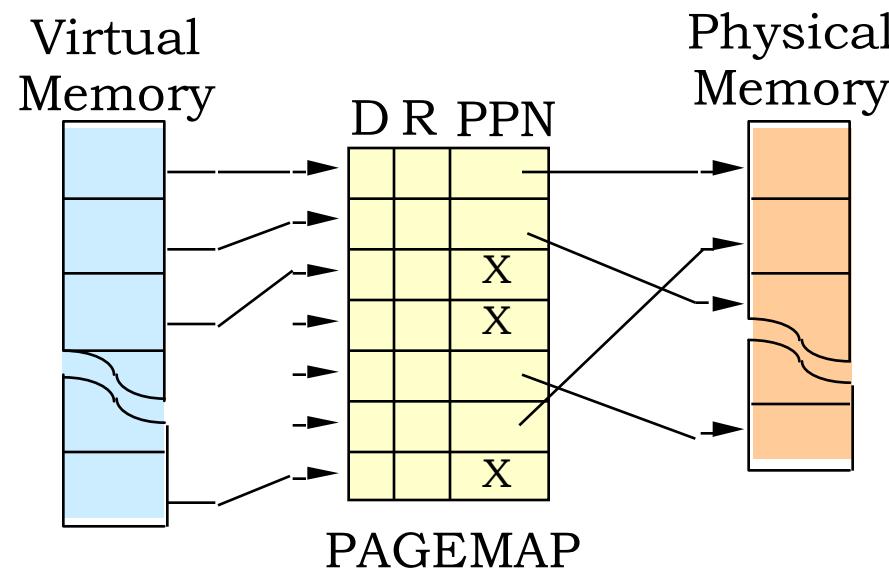
17. Virtualizing the Processor

6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

Recap: Virtual Memory

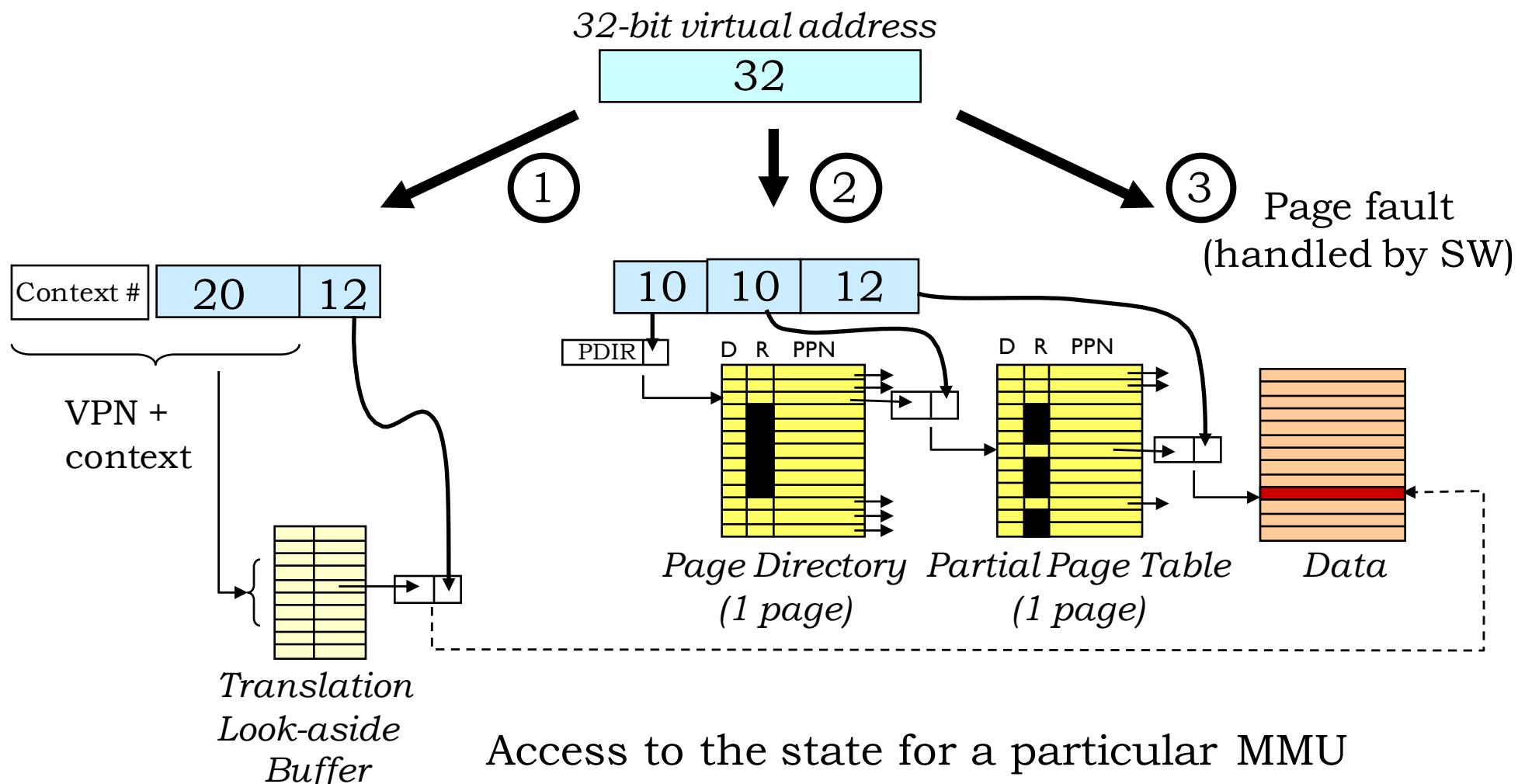
Review: Virtual Memory



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

MMU Address Translation

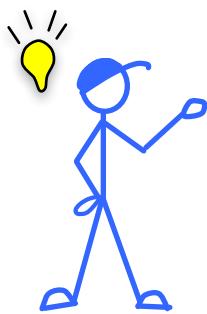


Access to the state for a particular MMU context is controlled by two registers:

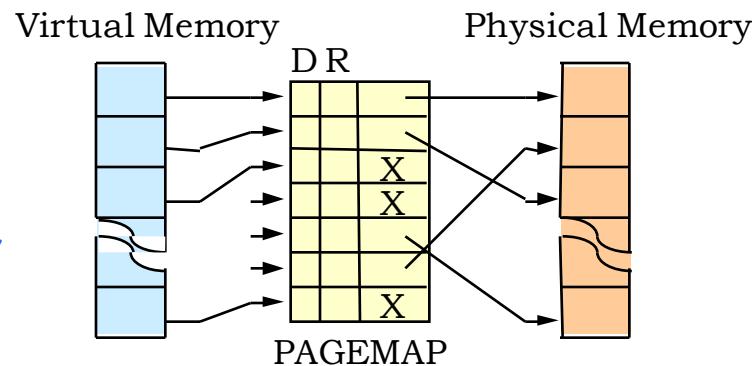
1. Context #, used by TLB
2. PDIR, used by multi-level page map

Contexts

A context is an entire set of mappings from VIRTUAL to PHYSICAL page numbers as specified by the contents of the page map:



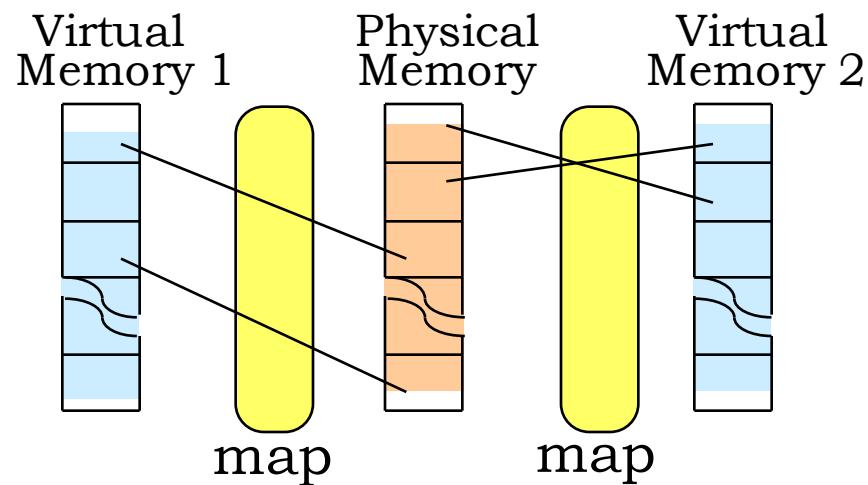
We would like to support multiple VIRTUAL to PHYSICAL Mappings and, thus, multiple Contexts.



THE BIG IDEA: Several programs, each with their own context, may be simultaneously loaded into main memory!

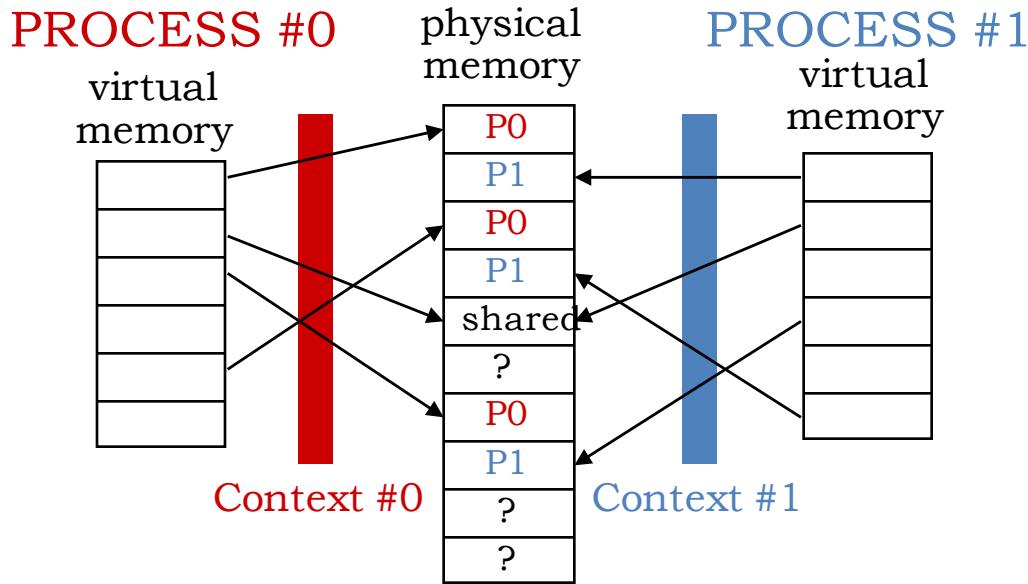
“Context switch”: reload the page map!

 We only have change Context# and PDIR



Processes

Building a Virtual Machine (VM)



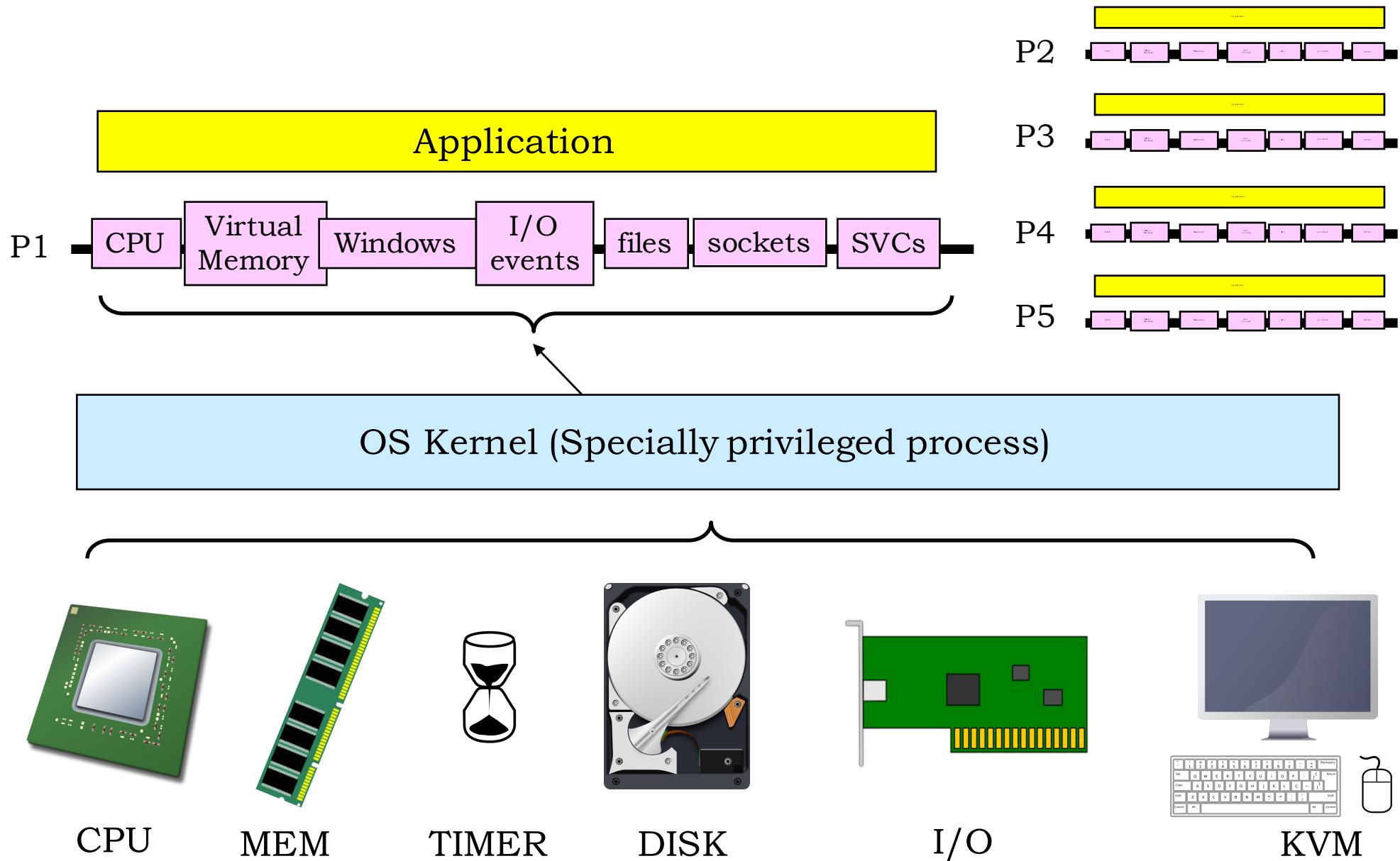
Goal: give each program its own “VIRTUAL MACHINE”; programs don’t “know” about each other...

New abstraction: a process which has its own

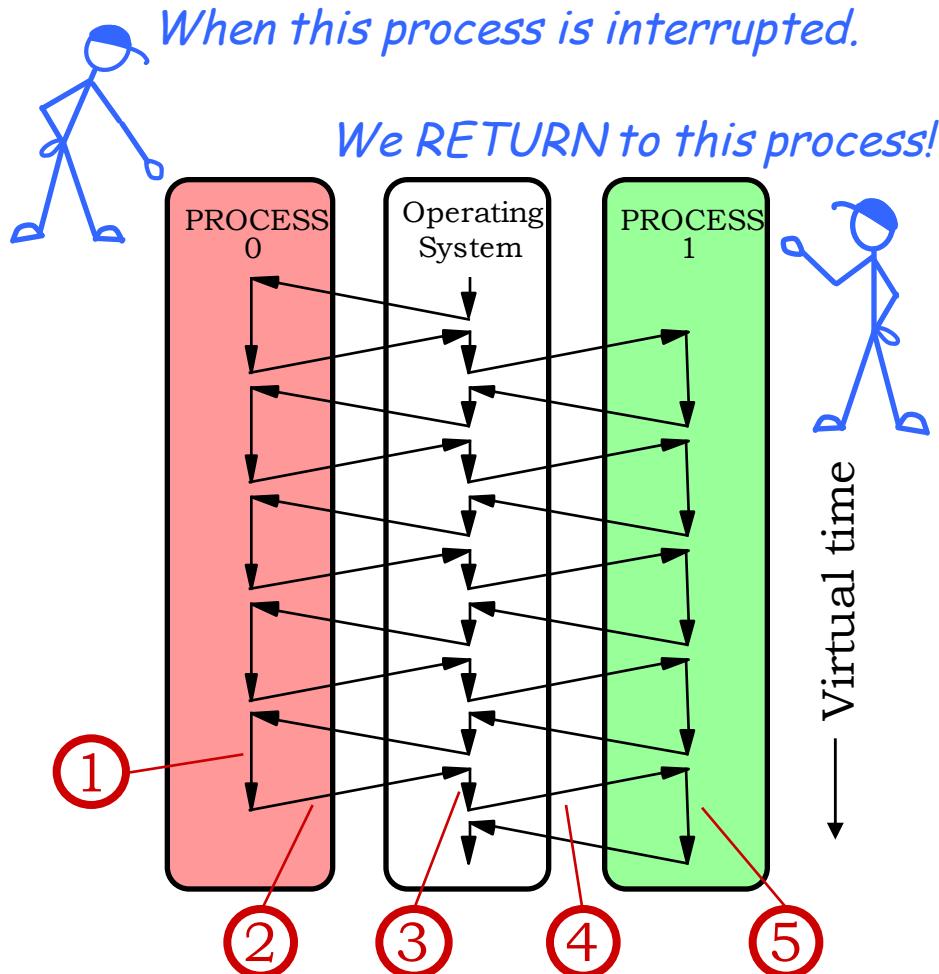
- machine state: R0, ..., R30
- context (virtual address space)
- PC, stack
- program (w/ shared code)
- virtual I/O devices

“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.

One VM For Each Process



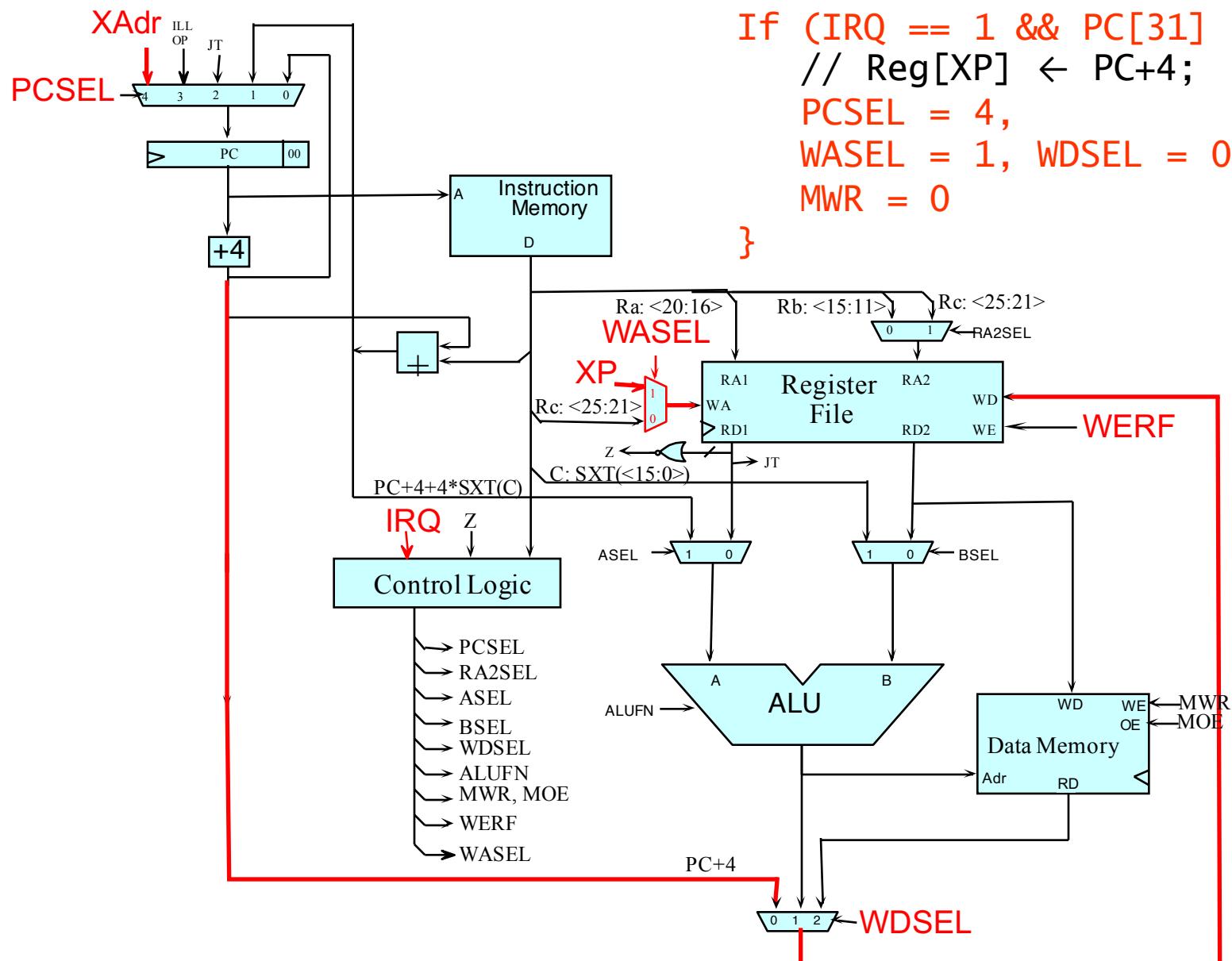
Processes: Multiplexing the CPU



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

Timesharing

Key Technology: Timer Interrupts



Beta Interrupt Handling

Minimal Hardware Implementation:

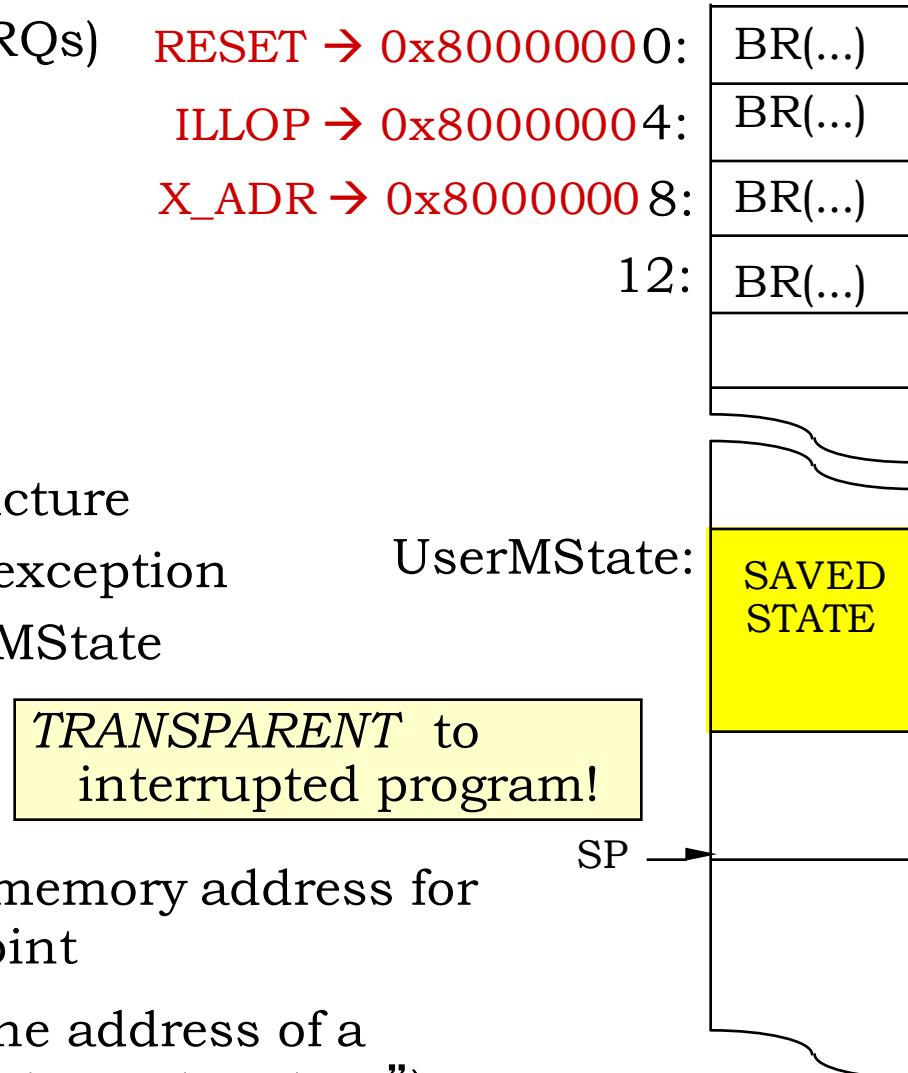
- Check for Interrupt Requests (IRQs) before each instruction fetch.
- On IRQ j:
 - copy PC+4 into Reg[XP];
 - INSTALL $j \times 4$ as new PC.

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”)



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”.

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();  
}
```

Handler
(written in C)

Clock_h:

```
ST(r0, UserMState)          // Save state of  
ST(r1, UserMState+4)        // interrupted  
...                          // app pgm...  
ST(r30, UserMState+30*4)  
LD(KStack, SP)              // Use KERNEL SP  
BR(Clock_Handler, 1p)        // 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.
```

“Interrupt stub”
(written in assy.)

Simple Timesharing Scheduler

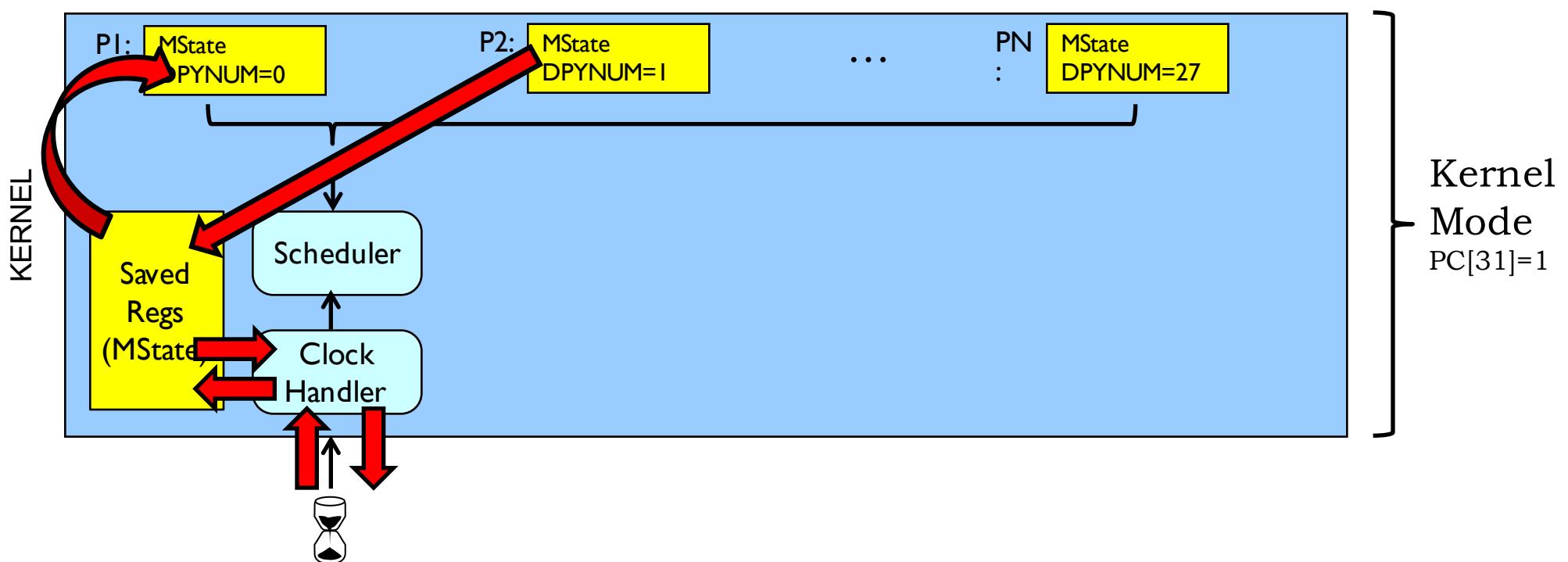
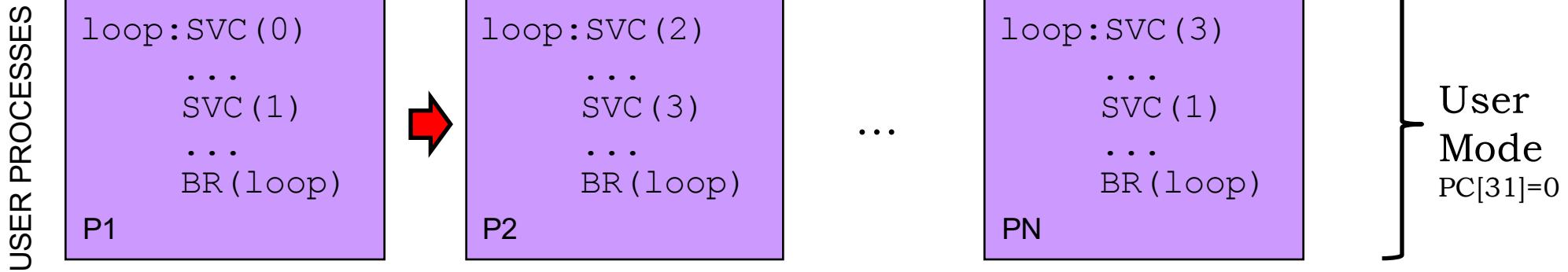
```
struct MState { int Regs[31]; } UserMState;
```

```
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
```

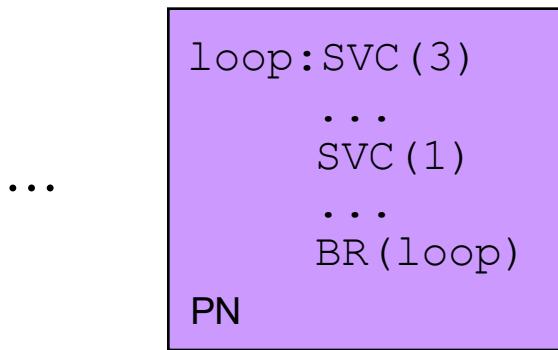
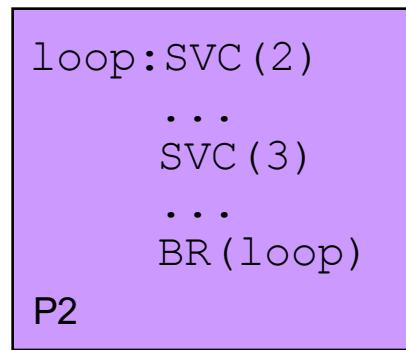
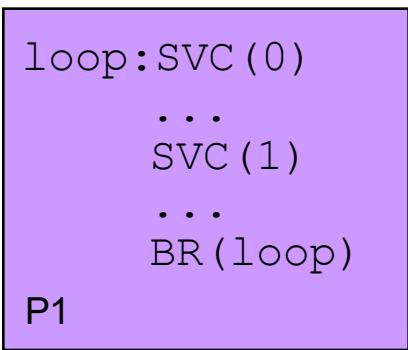
```
Scheduler() {
    ProcTbl[Cur].State = UserMState; // Save Cur state
    Cur = (Cur+1)%N;              // Incr mod N
    UserMState = ProcTbl[Cur].State; // Install state for next User
    LoadUserContext(ProcTbl[Cur].PageMap); // Install context
}
```

OS Organization: Processes



One Interrupt at a Time!

USER PROCESSES



Interrupts allowed!

User Mode
 $PC[31]=0$

KERNEL



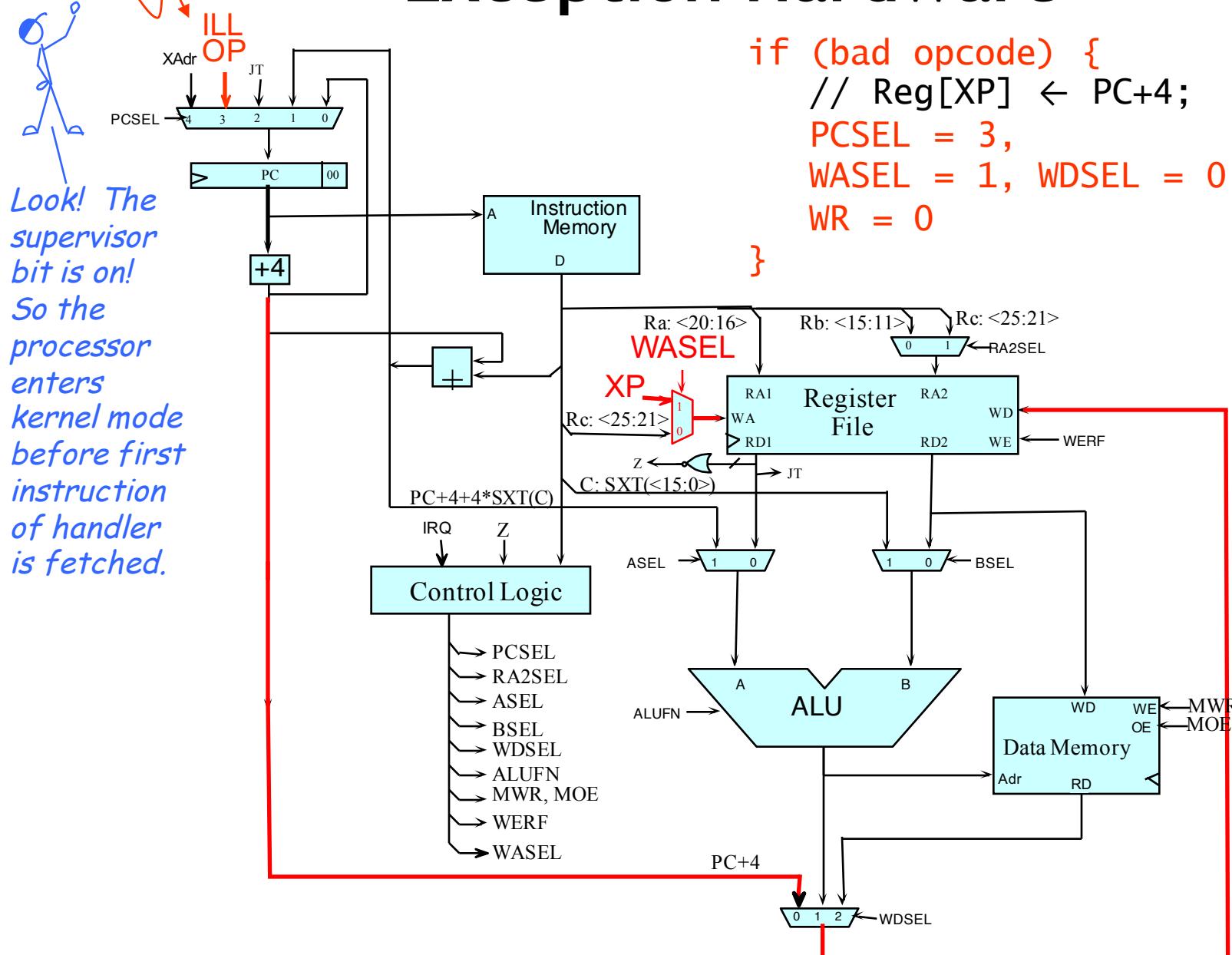
*Kernel code is
not re-entrant!*



No interrupts!

Handling Illegal Instructions

Exception Hardware



```

if (bad opcode) {
    // Reg[XP] < PC+4;  PC < "ILLop"
    PCSEL = 3,
    WASEL = 1, WDSEL = 0, WERF = 1,
    WR = 0
}

```

Exception Handling

```
// hardware interrupt vectors are in low memory
. = 0
    BR(I_Reset)      // when Beta first starts
    BR(I_Il10p)      // on Illegal Instruction (eg SVC)
    BR(I_C1k)         // 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
```



This is where the HW sets the PC during an illegal opcode exception

```
// 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
```

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*



Illegal Handler

```
/// Handler for Illegal Instructions
```

```
I_IlloP:
```

```
    save_all_regs(UserMState) // Save the machine state.  
    LD(KStack, SP)           // Install kernel stack pointer.
```

So kernel code can
make subroutine
calls!

```
    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)
```

A small blue stick figure pointing towards the text, labeled "supervisor bit...".

```
UUOTb1: BAD()
```

```
        UUO(SVC_UUO)
```

```
    BAD()
```

```
        BAD()
```

```
    BAD()
```

```
        BAD()
```

... more table follows ...

```
        UUO(swapreg)
```

```
    BAD()
```

This is a
64-entry
dispatch
table.
Each entry
is an
address of
a “handler”



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(r1)  
  
    ANDC(r0,0xFFFF,r1) // extract page offset  
    PUSH(r1)  
    SHRC(r0,12,r1)    // extract virtual page number  
    PUSH(r1)  
    BR(VtoP,LP)        // returns physical address in R0  
    DEALLOCATE(2)  
    LD(r0,0,r0)        // load the contents  
  
    POP(r1)           // restore regs, return to caller  
    POP(LP)  
    JMP(LP)
```

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.

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!
```

Supervisor Calls

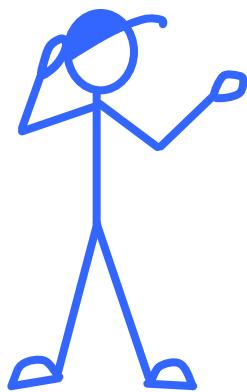
Communicating with the OS

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

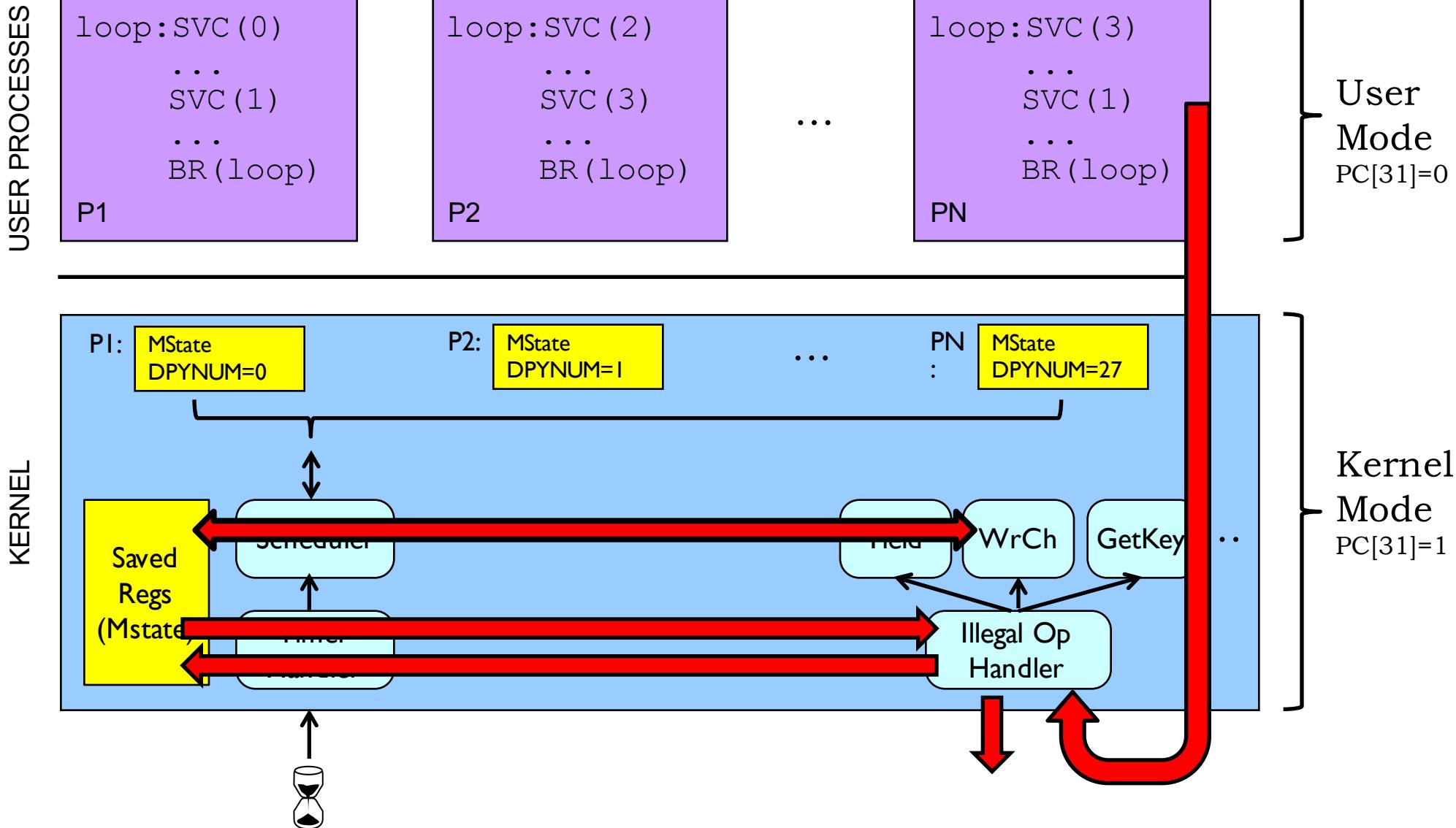
Implementation:

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!*

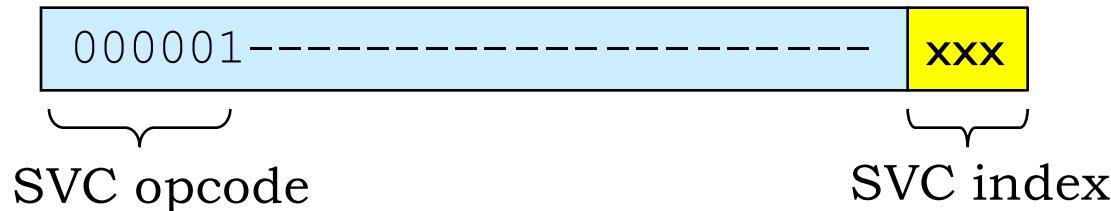


OS Organization: Supervisor Calls



Handler for SVCs

SVC Instruction format



// Sub-handler for SVCs, called from I_I11Op 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.  
JMP(r1)
```

SVCTb1: UU0(HaltH) // SVC(0): User-mode HALT instruction
UU0(WrMsgH) // SVC(1): Write message
UU0(WrChH) // SVC(2): Write Character
Another UU0(GetKeyH) // SVC(3): Get Key
dispatch UU0(HexPrtH) // SVC(4): Hex Print
table! UU0(WaitH) // SVC(5): Wait(S), S in R3
UU0(SignalH) // SVC(6): Signal(S), S in R3
UU0(YieldH) // SVC(7): Yield()

*Another
dispatch
table!*



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!
```

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_I11Op on SVC opcode:
// SVC instruction is in R0, it's address is Reg[XP]-4
```

```
SVC_UU0:
```

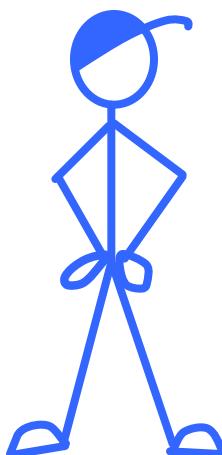
```
    ANDC(r0,0xF,r1)          // Pick out low bits,
    SHLC(r1,2,r1)            // make a word index,
    LD(r1,SVCTb1,r1)         // and fetch the table entry.
    JMP(r1)
```

```
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
```

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.

18. Devices and Interrupts

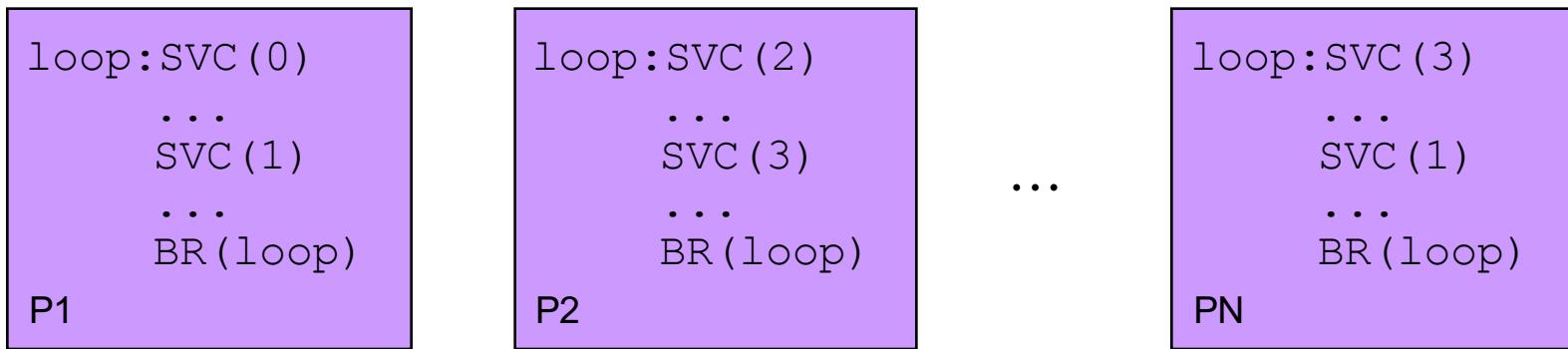
6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

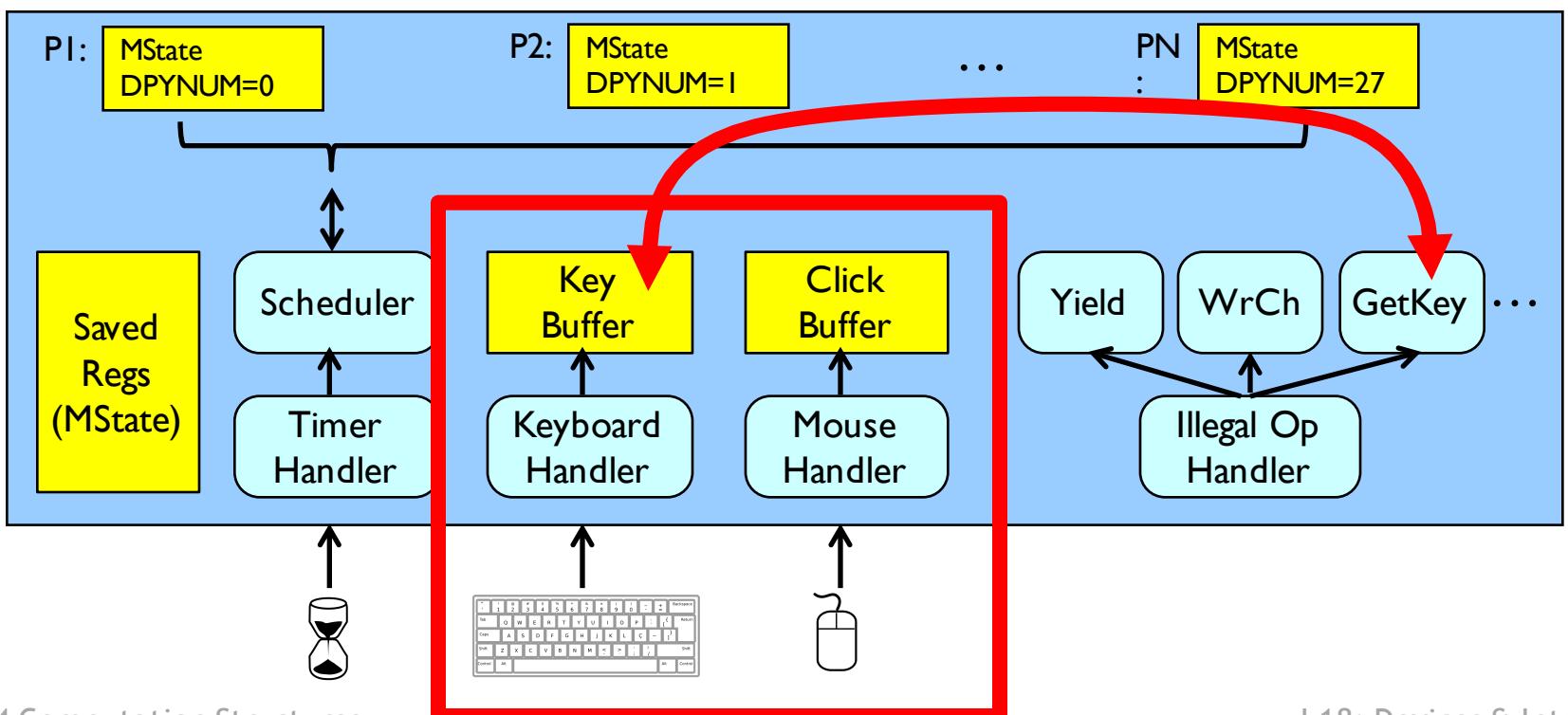
OS Device Handlers

OS Organization: I/O Devices

USER PROCESSES

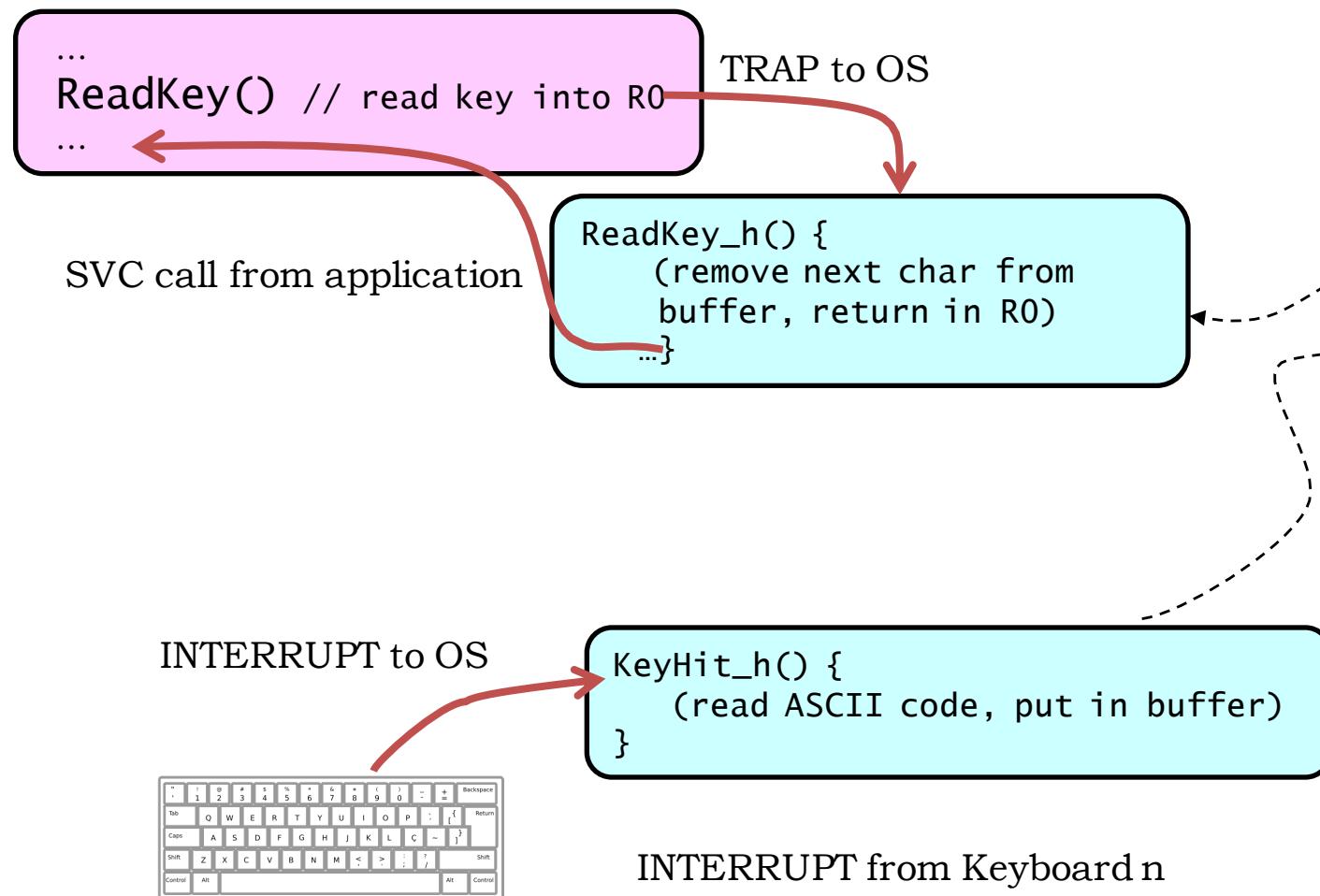


KERNEL



Asynchronous I/O Handling

Application:



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;  
}
```

SVCs for Input/Output

.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 = ProcTb1[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.

.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...

.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.Rgs[XP] = UserMState.Rgs[XP]-4;
        Scheduler();
    } else
        UserMState.Rgs[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.

Sophisticated Scheduling

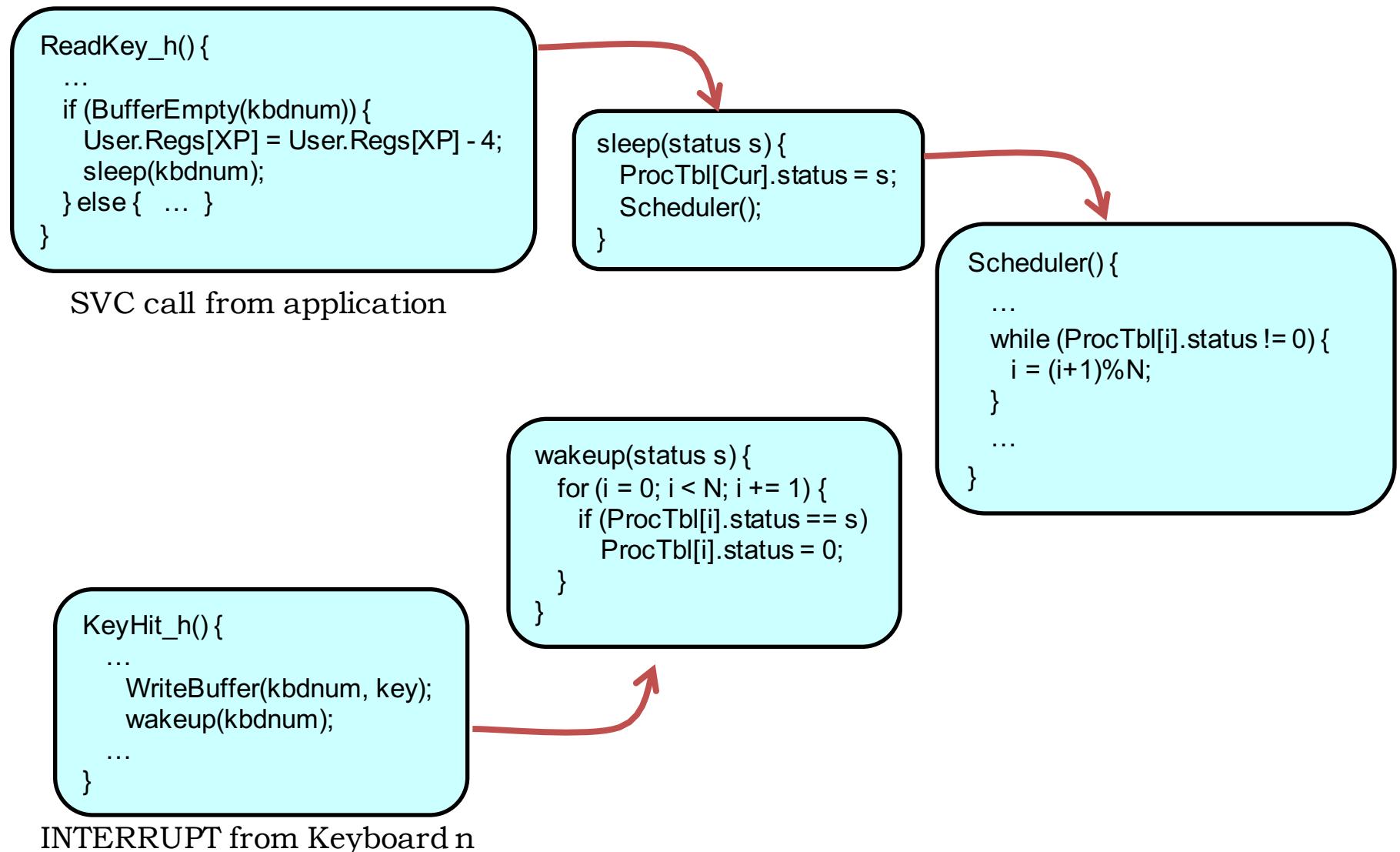
To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- 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:

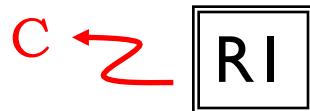
- `sleep(reason)` - Puts CurProc to sleep. “Reason” is an arbitrary binary value giving a condition for reactivation.
- `wakeup(reason)` - Makes active any process in `sleep(reason)`.

.ReadKey SVC: Attempt #4



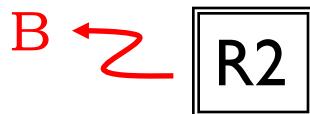
Example: Match Handler with OS

Example: Match Handler to OS



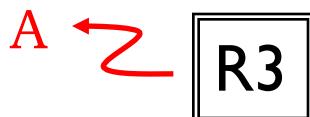
```
ReadCh_h() { // Version R1
    if (BufferEmpty(0))
        UserMState.Reg[XP] = UserMState.Reg[XP] - 4;
    else
        UserMState.Reg[0] = ReadInputBuffer(0);
}
```

Always reads from the same buffer



```
ReadCh_h() { // Version R2
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum));
    UserMState.Reg[0] = ReadInputBuffer(kbdnum);
}
```

Oops! Infinite loop?



```
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);
}
```

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

R1

```
ReadCh_h() { // Version R1
    if (BufferEmpty(0))
        UserMState.Rregs[XP] = UserMState.Rregs[XP] - 4;
    else
        UserMState.Rregs[0] = ReadInputBuffer(0);
}
```

R2

```
ReadCh_h() { // Version R2
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum));
    UserMState.Rregs[0] = ReadInputBuffer(kbdnum);
}
```

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);
}
```

	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!”

R1

```
ReadCh_h() { // Version R1
    if (BufferEmpty(0))
        UserMState.Rregs[XP] = UserMState.Rregs[XP] - 4;
    else
        UserMState.Rregs[0] = ReadInputBuffer(0);
}
```

R2

```
ReadCh_h() { // Version R2
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum));
    UserMState.Rregs[0] = ReadInputBuffer(kbdnum);
}
```

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);
}
```

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

	A	B	C
R1		X	X
R2	X	X	X
R3	X	X	X

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.”

R1

```
ReadCh_h() { // Version R1
    if (BufferEmpty(0))
        UserMState.Rregs[XP] = UserMState.Rregs[XP] - 4;
    else
        UserMState.Rregs[0] = ReadInputBuffer(0);
}
```

R2

```
ReadCh_h() { // Version R2
    int kbdnum = ProcTb1[Cur].DPYNum;
    while (BufferEmpty(kbdnum));
    UserMState.Rregs[0] = ReadInputBuffer(kbdnum);
}
```

R3

```
ReadCh_h() { // Version R3
    int kbdnum = ProcTb1[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        UserMState.Rregs[XP] = UserMState.Rregs[XP] - 4;
        Scheduler();
    } else
        UserMState.Rregs[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?

“Neat, the new system seems to work fine.
It even wastes less CPU time than it used to!”

Real Time

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

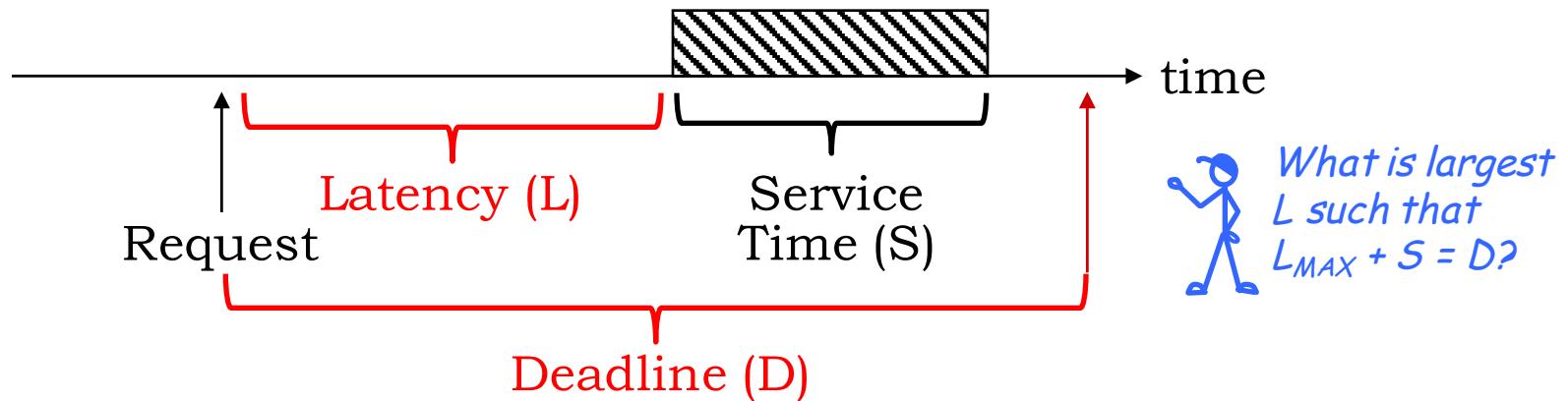
Real-time as an alternative to time-sliced
or fixed-priority preemptive scheduling



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?*



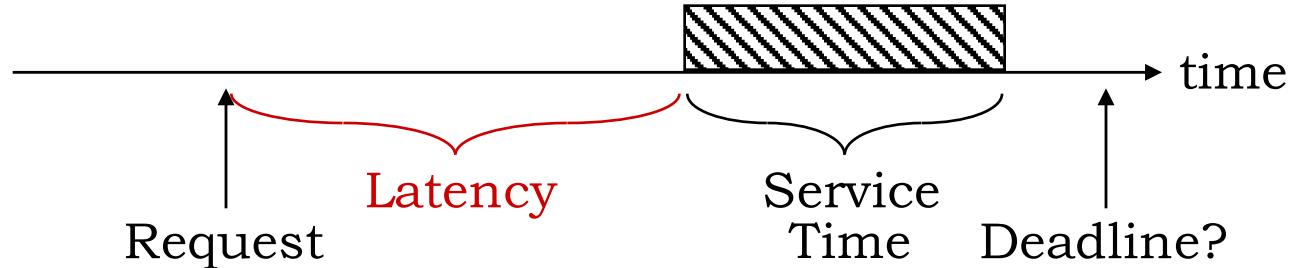
Sometimes bad things happen when service is delayed beyond its “dead”-line:

Missed characters
Automobile crashes
Nuclear meltdowns

“HARD”
Real time
constraints



Sources of Interrupt Latency



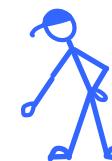
What causes interrupt latency:

- State save, context switch.
- Periods of un-interruptability:

*But, this is
application
dependent!*



*We can consider this
when we write our O/S*



*We can address
this in our ISA*

- Long, uninterruptable instructions – e.g. block moves
- Explicitly disabled periods (e.g. during service of other interrupts).

GOAL: BOUND (and minimize) interrupt latency!

- Optimize interrupt sequence context switch
- Make unbounded-time instructions *interruptable* (state in registers, etc).
- Avoid/minimize disable time
- Allow handlers to be interrupted, in certain cases.

Weak Priorities

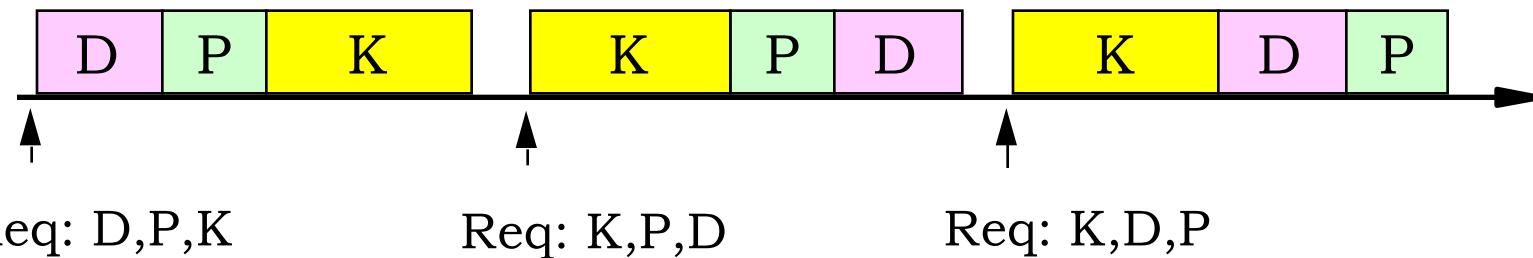
Scheduling of Multiple Devices

"TOY" System scenario:

<u>Actual w/c Latency</u>	<u>DEVICE</u>	<u>Service Time</u>
$500 + 400 = \underline{900}$	Keyboard	800
$800 + 400 = \underline{1200}$	Disk	500
$800 + 500 = \underline{1300}$	Printer	400



What is the WORST CASE latency seen by each device?

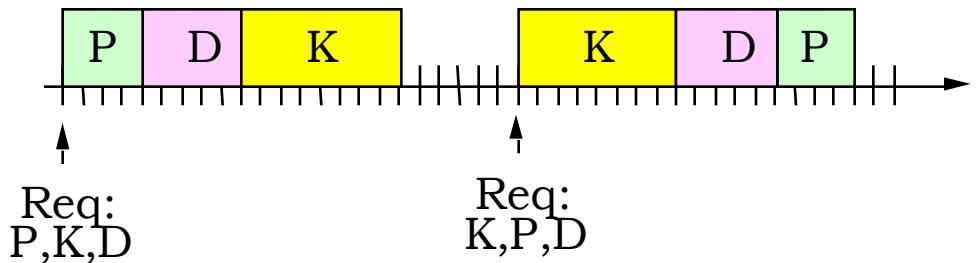


Assumptions:

- 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!

Weak (Non-preemptive) Priorities

ISSUE: Processor becomes interruptable 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	<u>DEVICE</u>	Service Time
<u>900</u>	Keyboard	800
<u>800</u>	Disk	500
<u>1300</u>	Printer	400

vs 1200 –
Now delayed by only 1 service!

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.

Strong Priorities

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!

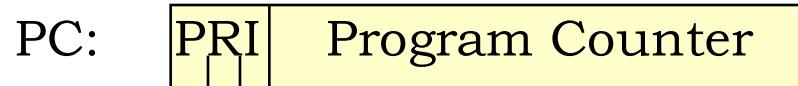
need **PREEMPTION**: Allow handlers for LOWER PRIORITY interrupts to be interrupted by HIGHER priority requests!

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 $P_{DEV} > \text{PRI}$, set $\text{PRI} = P_{DEV}$.



Strong priorities:

KEY: Priority in Processor state

Allows interruption of (certain) handlers

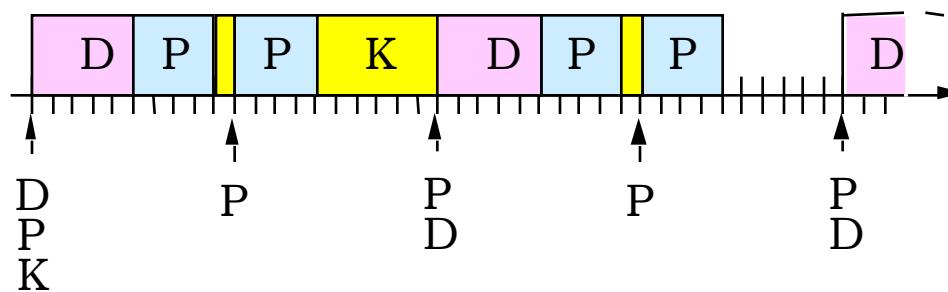
Allows preemption, but not reentrance

BENEFIT: Latency seen at high priorities UNAFFECTED by service times at low priorities.

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

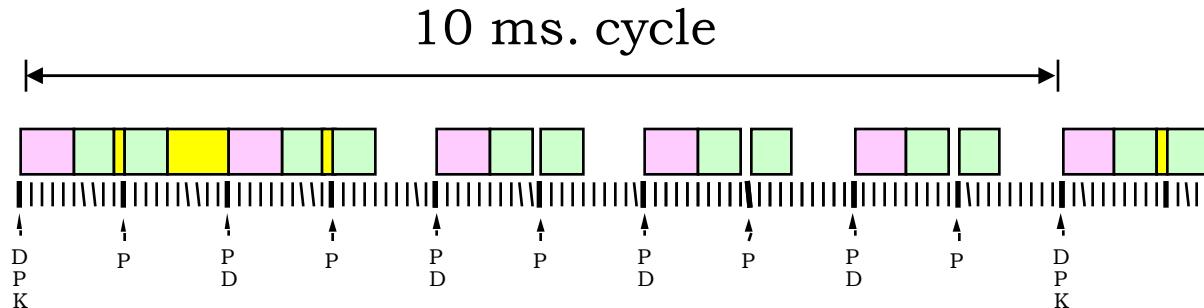


Note that interrupt LATENCIES don't tell the whole story—consider COMPLETION TIMES, e.g., for Keyboard in the example above.

Keyboard service not complete until 3 ms after request!

Interrupt Load

How much CPU time is consumed by interrupt service?

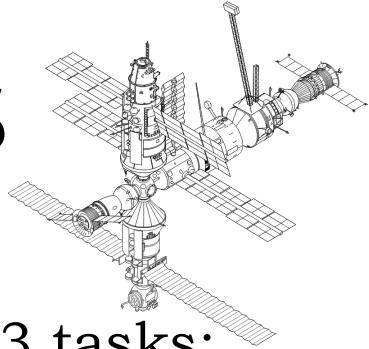


P	Latency	Device	Service Time (S)	Deadline (D)	L_{MAX}	Max Freq.	% Load
I	900us	Keyboard	800us			100/s	$800\text{us} * 100/\text{s} = 8\%$
3	0	Disk	500us	800us	300us	500/s	$500\text{us} * 500/\text{s} = 25\%$
2	500us	Printer	400us			1000/s	$400\text{us} * 1000/\text{s} = 40\%$

- 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

Example: Priorities in Action!

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 $G > \text{SSG} > \text{CP}$
- What fraction of the time will the processor spend idle? 48.33%
- What is the worst-case completion time for each task?

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.

	<i>Task</i>	<i>Period</i>	<i>Service time</i>	<i>Deadline</i>	
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? $100 - (3 \cdot 10) - (4 \cdot 5) = 50$
2. What fraction of the time will the processor spend idle? 8.33%
3. What is the worst-case completion time for each task?

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

19. Concurrency & Synchronization

6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

Interprocess Communication

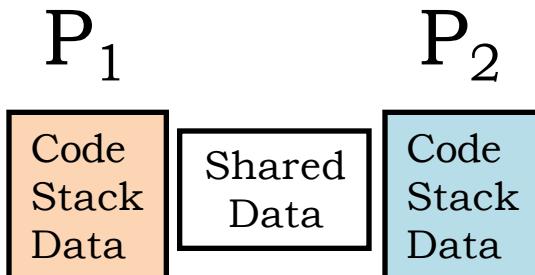
Interprocess Communication

Why have multiple processes?

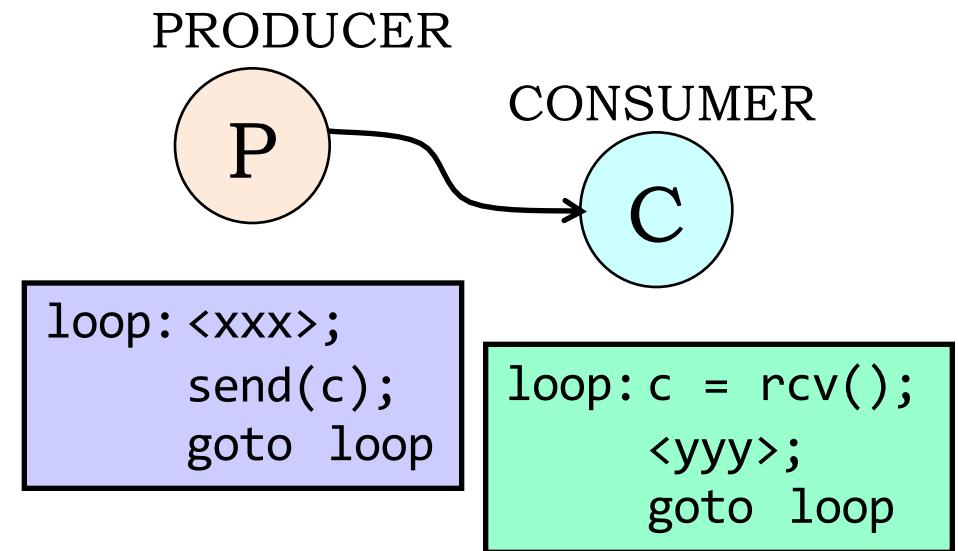
- Concurrency
- Asynchrony
- Processes as a programming primitive
- Data-/Event-driven

How to communicate?

- Shared Memory
(overlapping contexts)...
- Synchronization instructions
(hardware support)
- Supervisor calls



Classic Example:
“Producer-Consumer” Problem

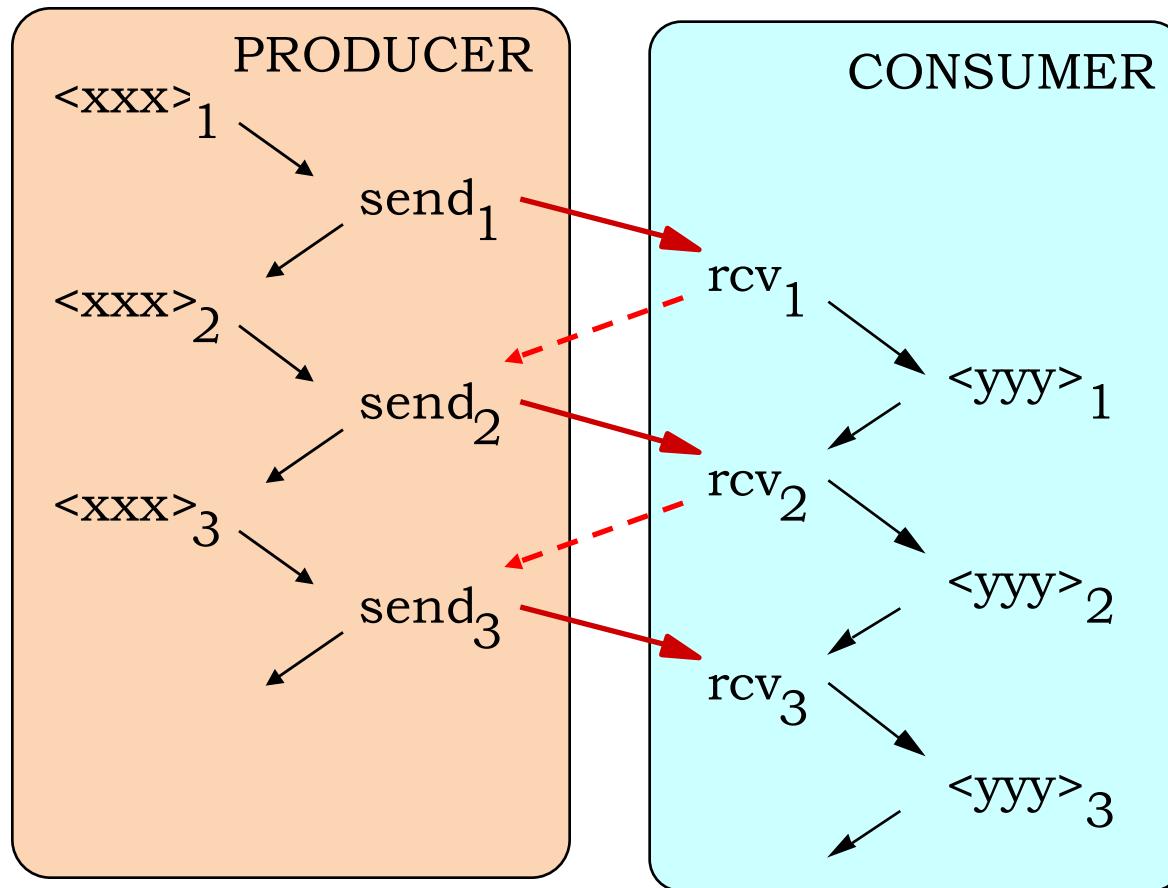


Real-World Examples:
Compiler/Assembler,
Application Frontend/Backend,
UNIX pipeline

Synchronous Communication

```
loop: <xxx>;  
      send(c);  
      goto loop
```

```
loop: c = recv();  
<yyy>;  
      goto loop
```



Precedence
Constraints:

$$a < b$$

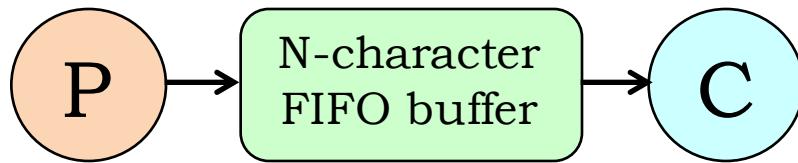
“a precedes b”

- Can't consume data before it's produced
- Producer can't “overwrite” data before it's consumed

$$\text{send}_i < \text{recv}_i$$

$$\text{recv}_i < \text{send}_{i+1}$$

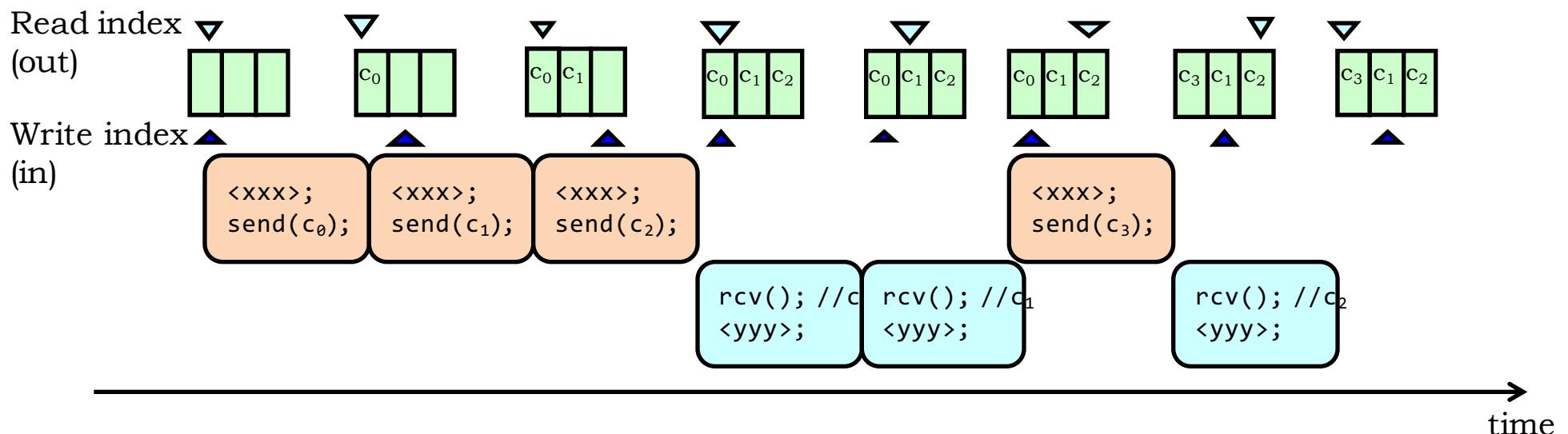
FIFO Buffering



RELAXES interprocess synchronization constraints.
Buffering relaxes the following OVERWRITE constraint to:

$$rcv_i < send_{i+N}$$

“Circular 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;  
}
```

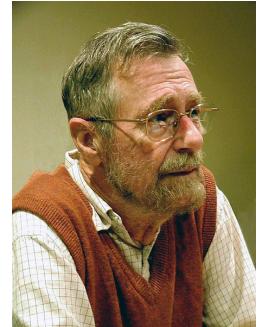
CONSUMER:

```
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()`)

Semaphores

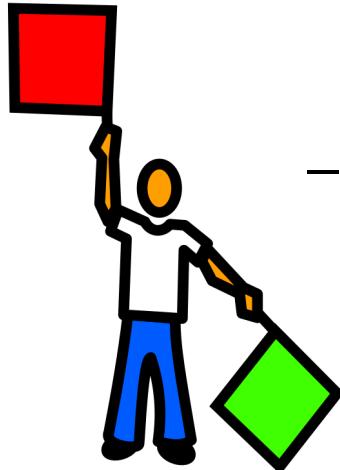
Semaphores (Dijkstra)



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 $s = s - 1$

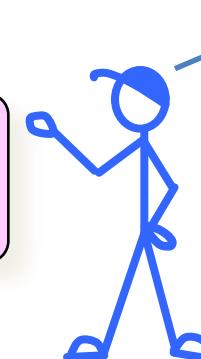
- `signal(semaphore s)`

$s = s + 1$ (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)

$\text{signal}(s)_i \prec \text{wait}(s)_{i+K}$



This is a precedence relationship: the i^{th} call to signal must complete before the $(i+K)^{\text{th}}$ call to wait will succeed.

Semaphores for Precedence

```
semaphore s = 0;
```

Process A

Process B

A1;

B1;

A2;

B2;

```
signal(s);
```

A3;

B3;

A4;

B4;

A5;

B5;



Goal: want statement A2 in process A to complete before statement B4 in Process B begins.

A2 < B4

Recipe:

- Declare semaphore = 0
- signal(s) at start of arrow
- wait(s) at end of arrow

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

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;  
}
```

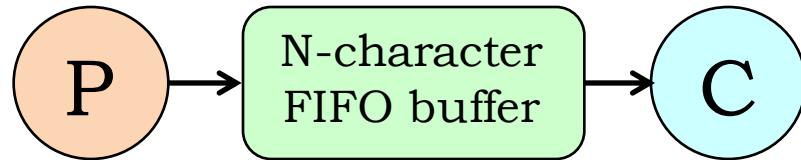
Does
this
work?



PRECEDENCE managed by semaphore: $\text{send}_i < \text{rcv}_i$

RESOURCE managed by semaphore chars: # of chars in buf

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:

$$\text{send}_i < \text{recv}_i$$

WHAT we still need:

$$\text{recv}_i < \text{send}_{i+N}$$

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:

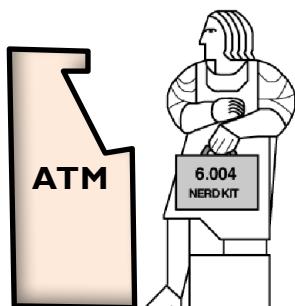
```
char rcv()
{
    char c;
    wait(chars);
    c = buf[out];
    out = (out+1)%N;
    signal(space);
    return c;
}
```

Resources managed by semaphore: characters in FIFO,
spaces in FIFO. Works with single producer, consumer. But
what about multiple producers and consumers?

Atomic Transactions

Simultaneous Transactions

Suppose you and your friend visit the ATM at exactly the same time, and remove \$50 from your account. What happens?



Withdraw(6004, 50)



Withdraw(6004, 50)

```
Debit(int account, int amount) {  
    t = balance[account];  
    balance[account] = t - amount;  
}
```

What is *supposed* to happen?

<u>Process # 1</u>	<u>Process #2</u>
LD(R10, balance, R0)	LD(R10, balance, R0)
SUB(R0, R1, R0)	SUB(R0, R1, R0)
ST(R0, balance, R10)	ST(R0, balance, R10)
...	...

NET: You have \$100, and your bank balance is \$100 less.

But, What If...

Process # 1

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!



Process #2

We need to be careful when writing concurrent programs. In particular, when modifying shared data.

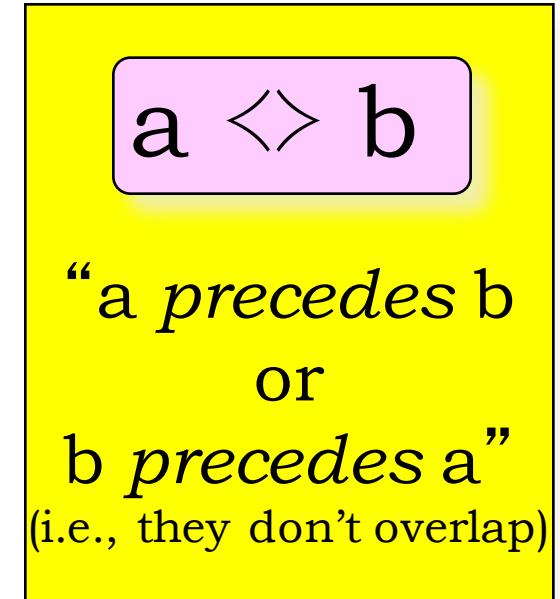
For certain code segments, called **CRITICAL SECTIONS**, we would like to ensure that no two executions overlap.

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.

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

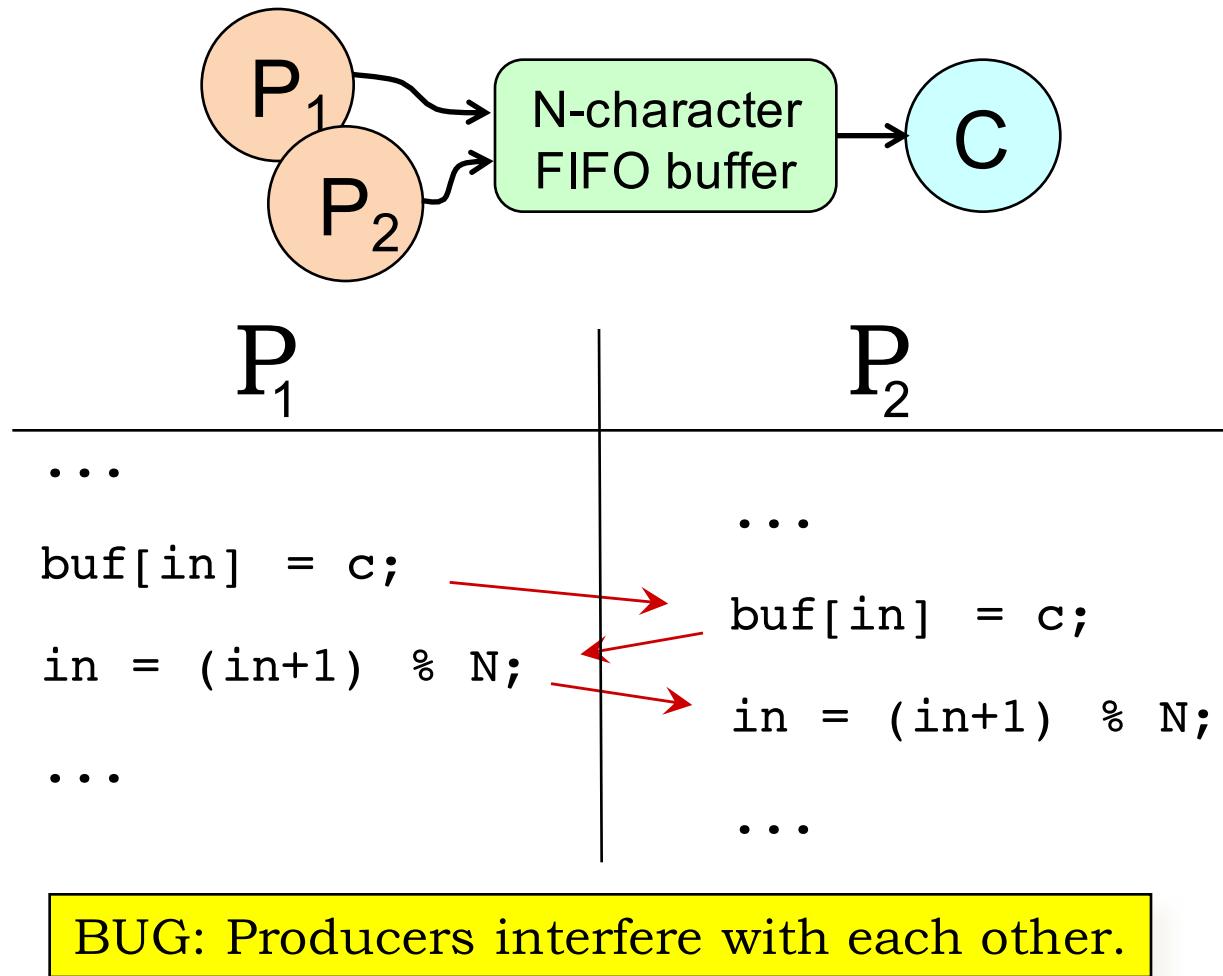
- 1 lock for whole balance database?
- 1 lock per account?
- 1 lock for all accounts ending in 004



Look up "database" on Wikipedia to learn about systems that support efficient transactions on shared data.

Producer/Consumer Atomicity Problems

Consider multiple PRODUCER processes:



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;
}
```

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:

$$\begin{aligned} \text{send}_i &\prec \text{rcv}_i \\ \text{rcv}_i &\prec \text{send}_{i+N} \end{aligned}$$

Mutual-exclusion relationships:

protect variables
in and *out*

Semaphore Implementation

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 uninterruptable 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.



Semaphores as a Supervisor Call

```
wait_h( ) {  
    int *addr;  
    addr = VtoP(User.Reg[0]);      // 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[0]);      // 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.

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)  
       BEQ(R0,wait)  
       ... critical section ...  
       CMOVE(1,R0)  
       ST(R0, lock, R31) }
```

} wait(lock)

```
          ST(R0, lock, R31) }
```

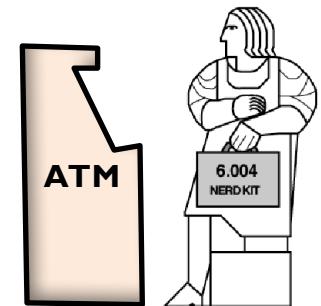
} signal(lock)

Deadlock

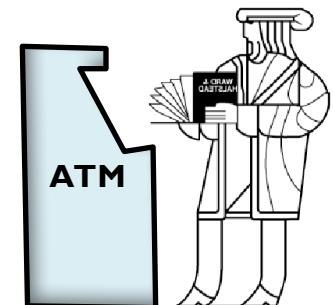
Synchronization: The Dark Side

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]);  
}
```



Transfer(6005, 6004, 50)



Transfer(6004, 6005, 50)

DEADLOCK (aka “deadly embrace”)!

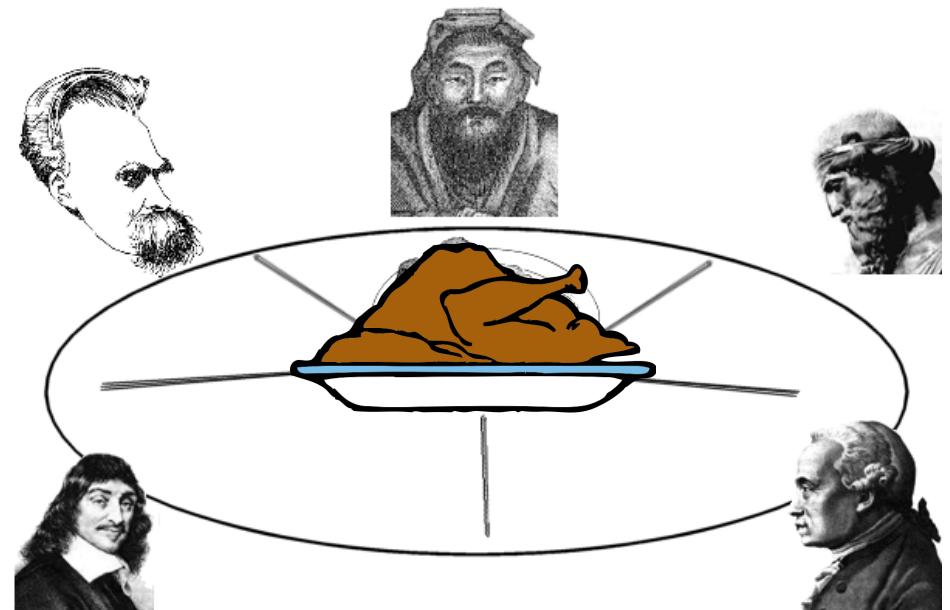
Dining Philosophers

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



*Wait, I think I see a
problem here... Shut up!!*



Deadlock!

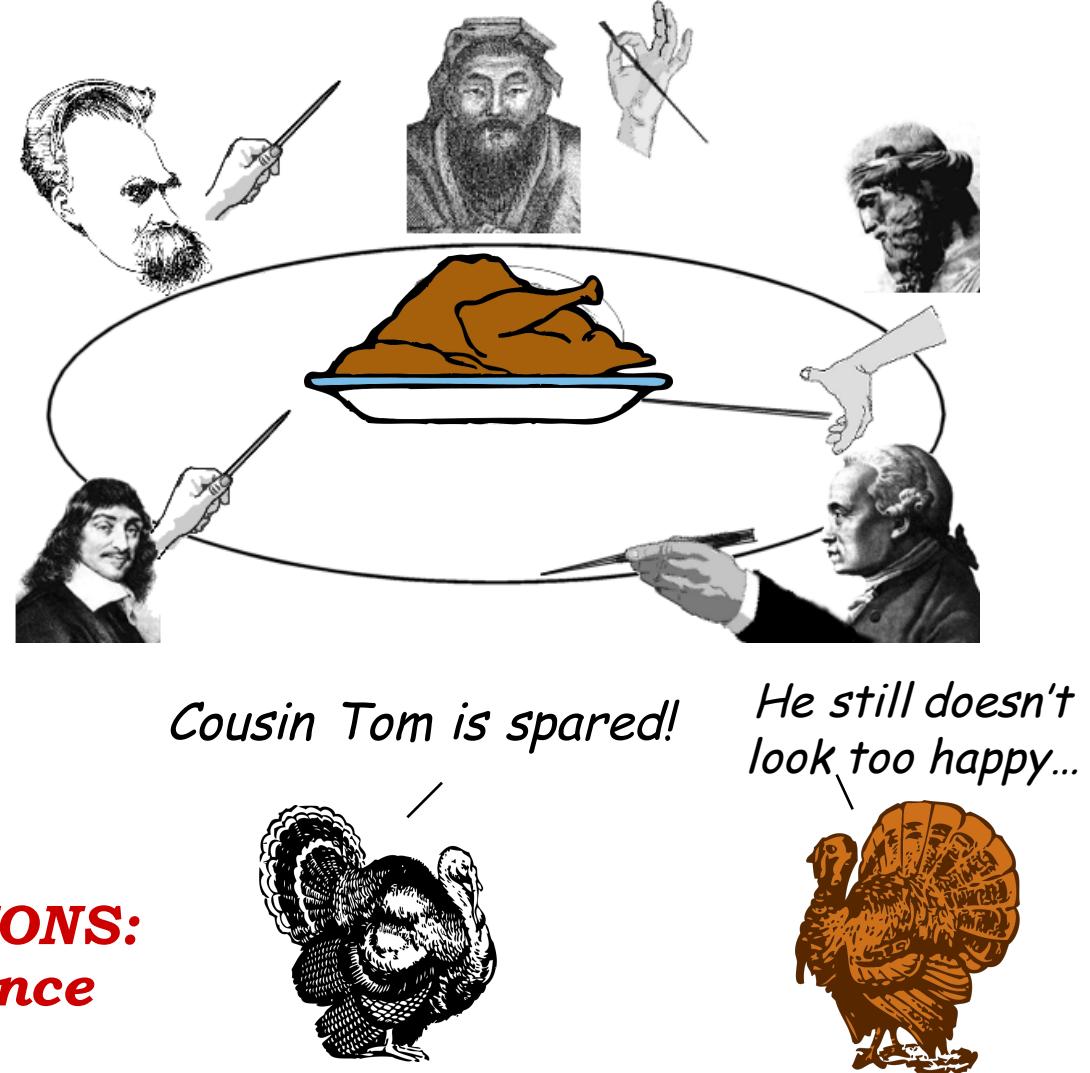
No one can make progress because they are all waiting for an unavailable resource

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

SOLUTIONS:
Avoidance
-or-

Detection and Recovery



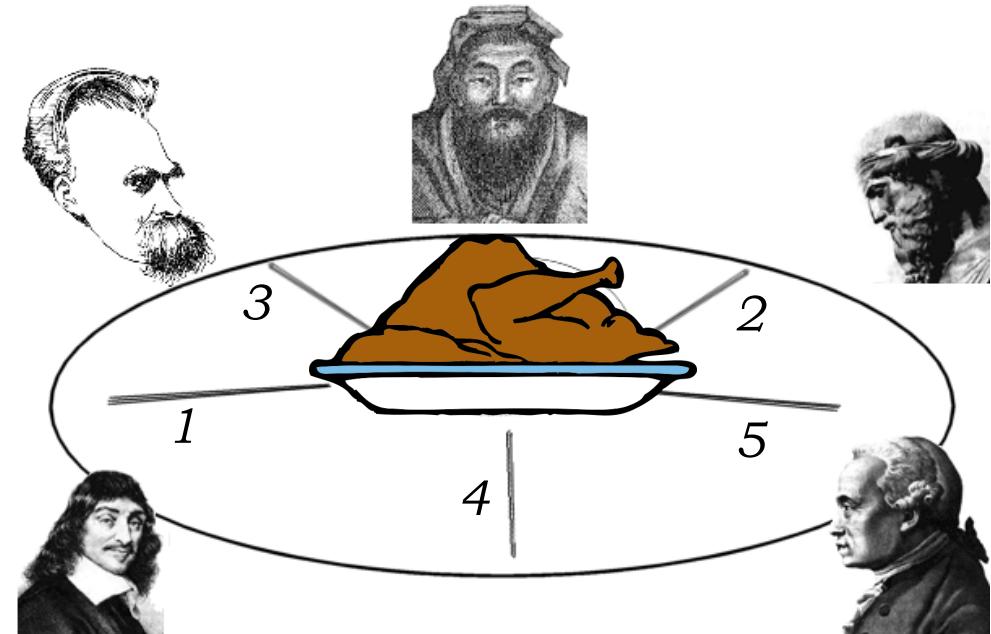
One Solution

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.

Dealing With Deadlocks

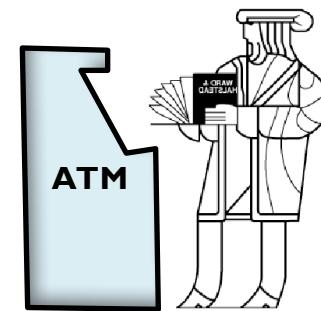
Cooperating processes:

- 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

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.

20. System-level Communication

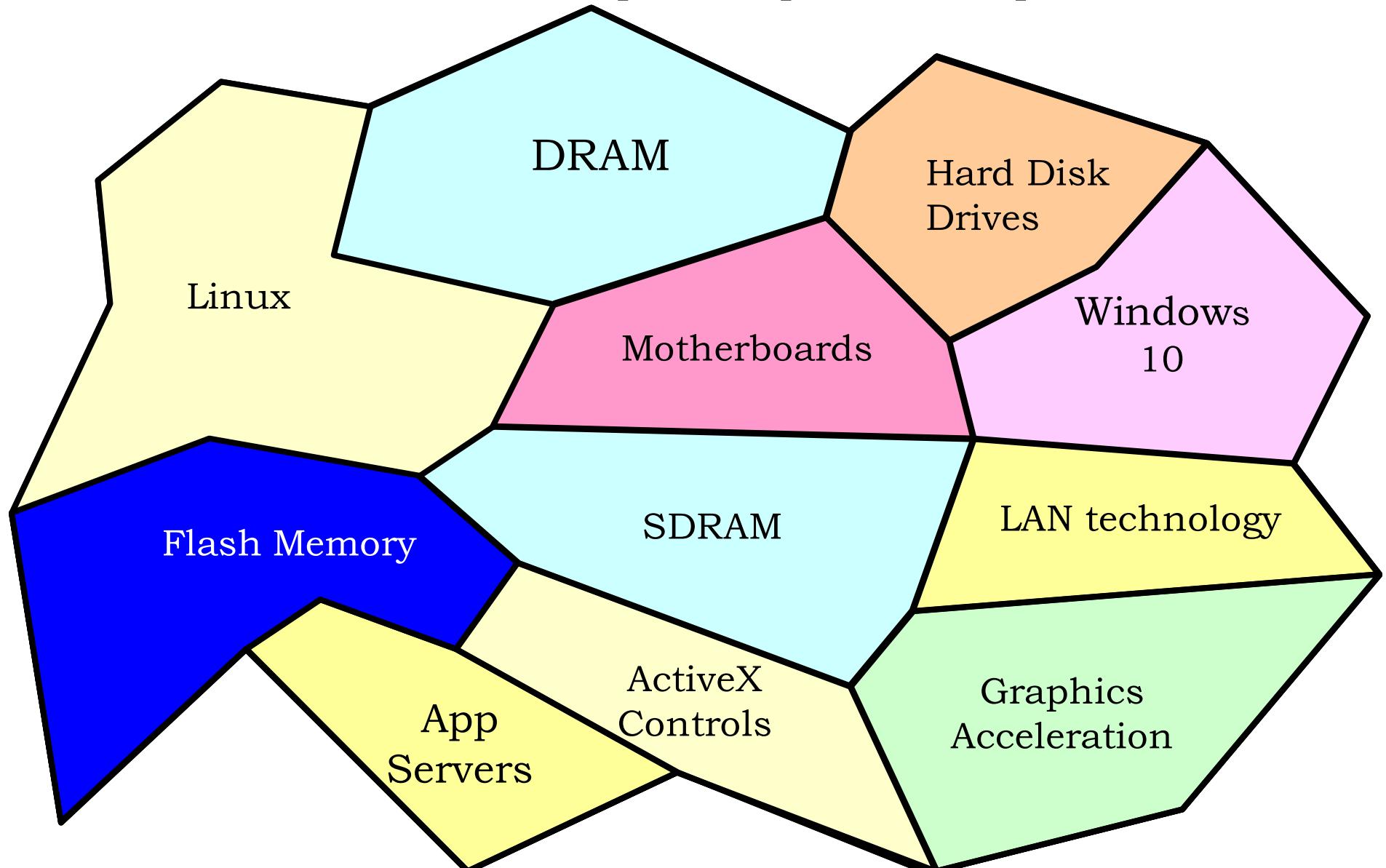
6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

System-level Interfaces

Computer System Technologies

What is the most important part of this picture?

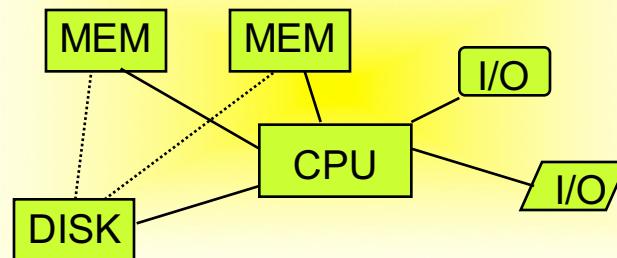


Technology comes & goes; interfaces last forever

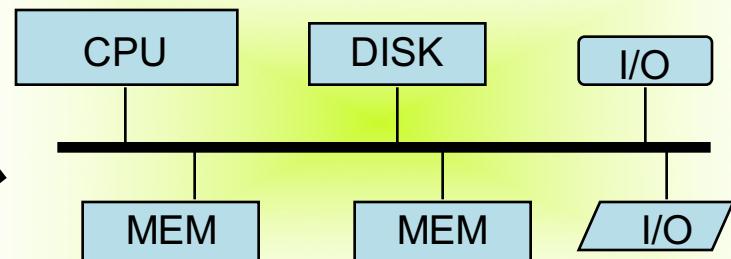
- Interfaces typically deserve more engineering attention than the technologies they interface...
 - Abstraction: should outlast many technology generations
 - Often “virtualized” to extend beyond original function (e.g. memory, I/O, services, machines)
 - Represent more potential value to their proprietors 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

System Interfaces & Modularity

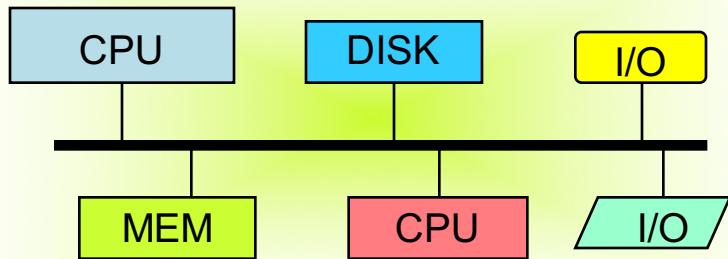
Ancient Times (Ad hoc connections)



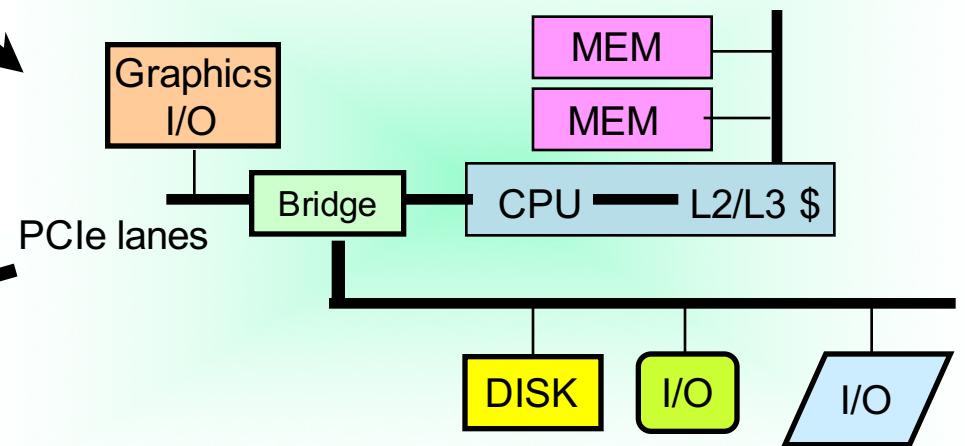
Late 60s (Processor-dependent Bus)



80s (Processor-independent Bus)



90s (buses galore)



(Mostly) Point-to-point

Wires

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.

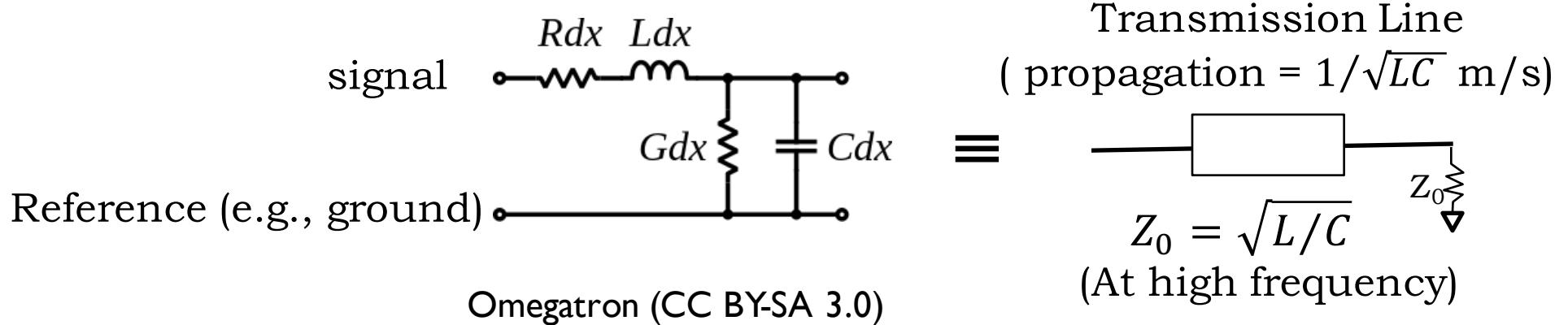
Distance matters.

Time matters.

Reality matters.



Electrical Model for Real Wires



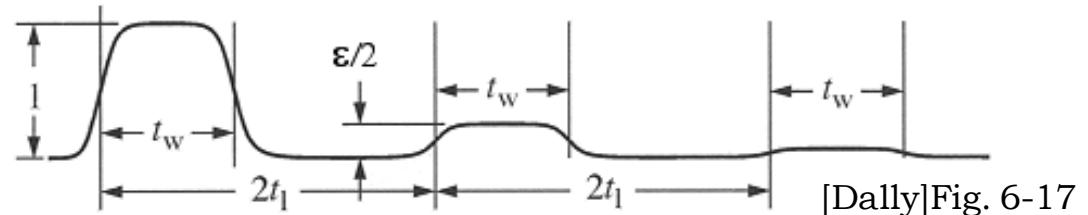
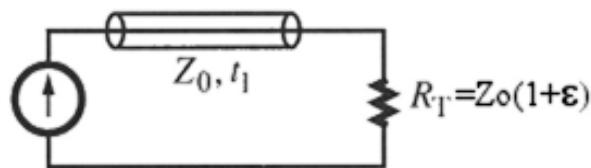
	Description	On chip	On PCB
R	Resistance of conductor	150k Ω /m	5 Ω /m
L	Self-inductance of conductor (due to magnetic field induced by current)	600nH/m	300nH/m
C	Capacitance between signal and ground	200pF/m	100pF/m
G	Conductance between signal and ground (through insulator)	small	small

http://cva.stanford.edu/books/dig_sys_engr/lectures/

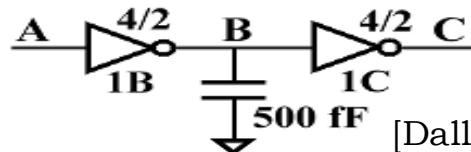
Real-World Consequences

ΔV from energy storage left over from earlier signaling on the wire:

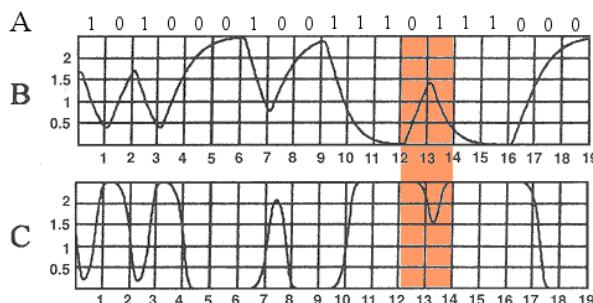
- transmission line discontinuities
(reflections off of impedance mismatches and terminations)



- charge storage in RC circuit
(narrow pulses are lost due to incomplete transitions)

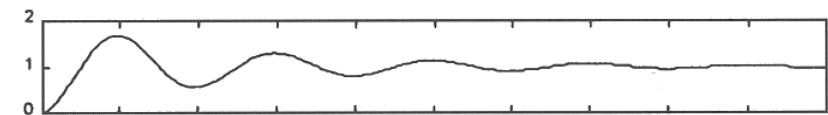
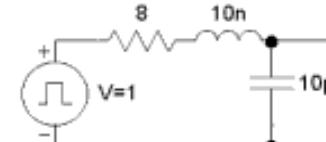


[Dally] Fig. 6-19



[Dally] Fig. 6-20

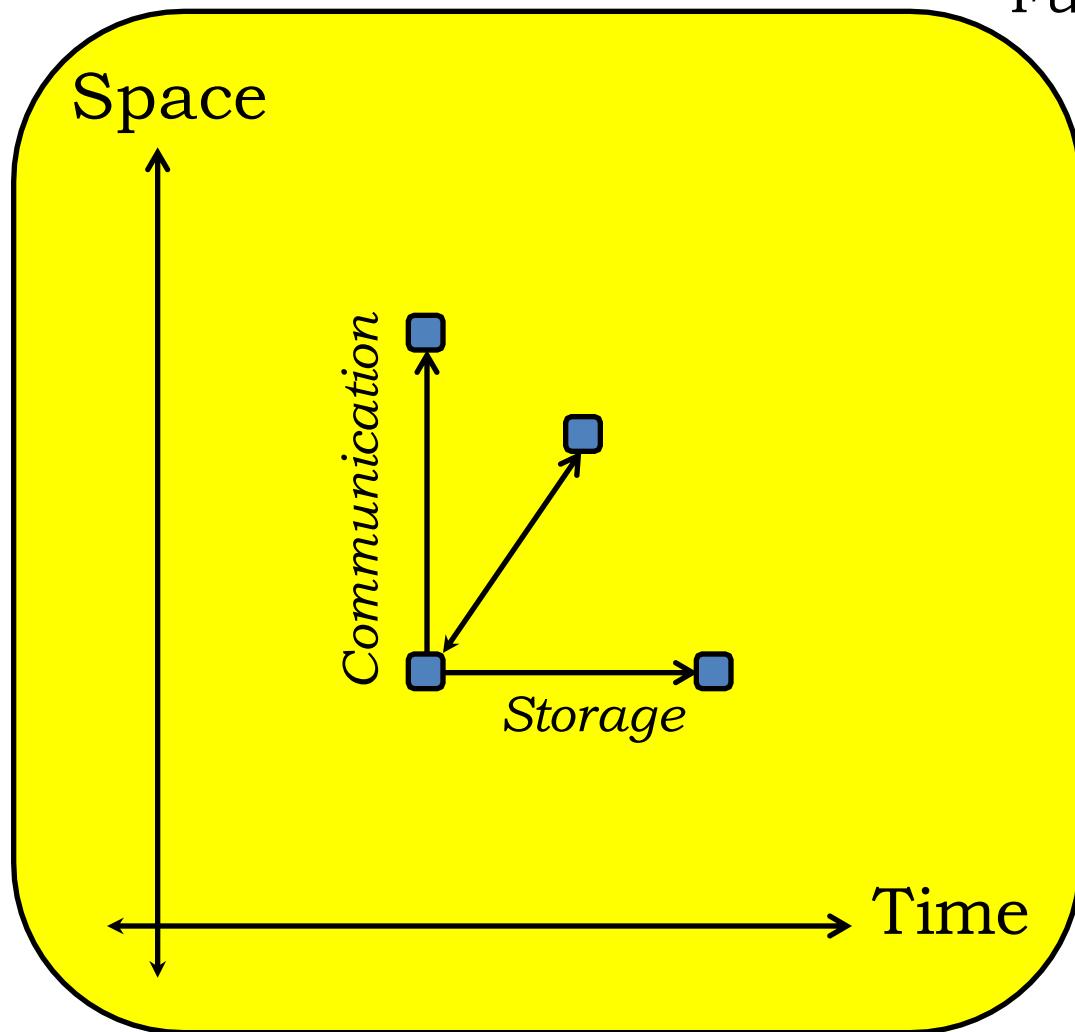
- RLC ringing (triggered by voltage steps)



Fix: slower operation, limiting voltage swings and slew rates

Dally, W.J., Poulton, J.W., *Digital Systems Engineering*, 1998

Space & Time Constraints



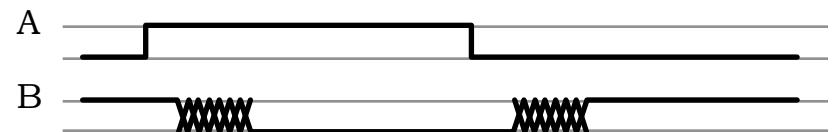
Fundamental Physical Constraints:

- 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

Gates, Wires, & Delays

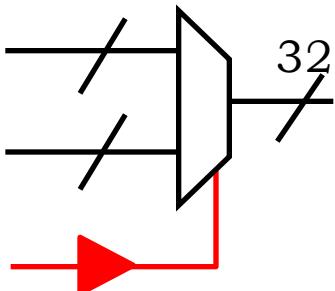
Our t_{pd} , t_{cd} timing model

- 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!).



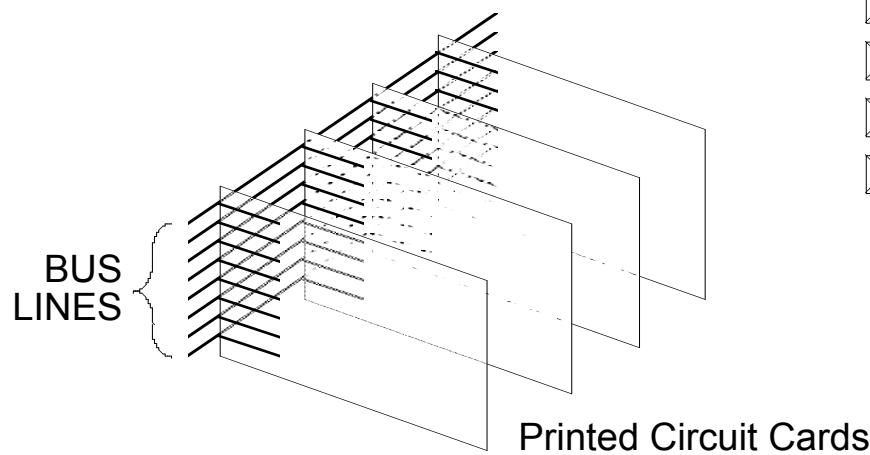
Particularly problematic: system-wide interconnect!

Buses

Interface Standard: Backplane Bus

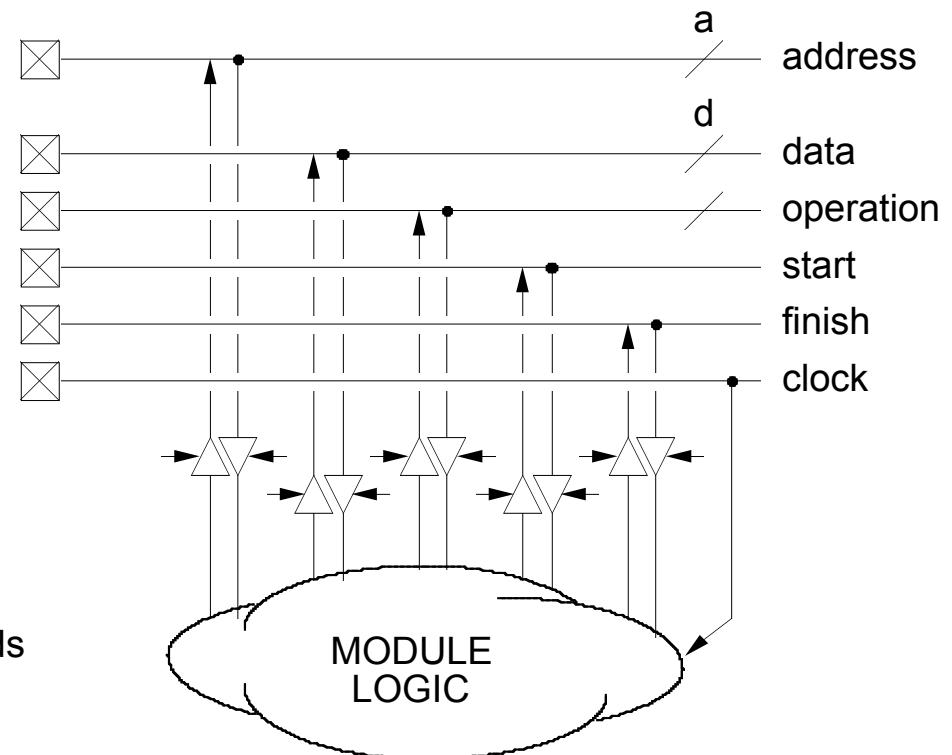
Modular cards that plug into a common backplane:

- CPUs
- Memories
- Bulk storage
- I/O devices
- S/W?

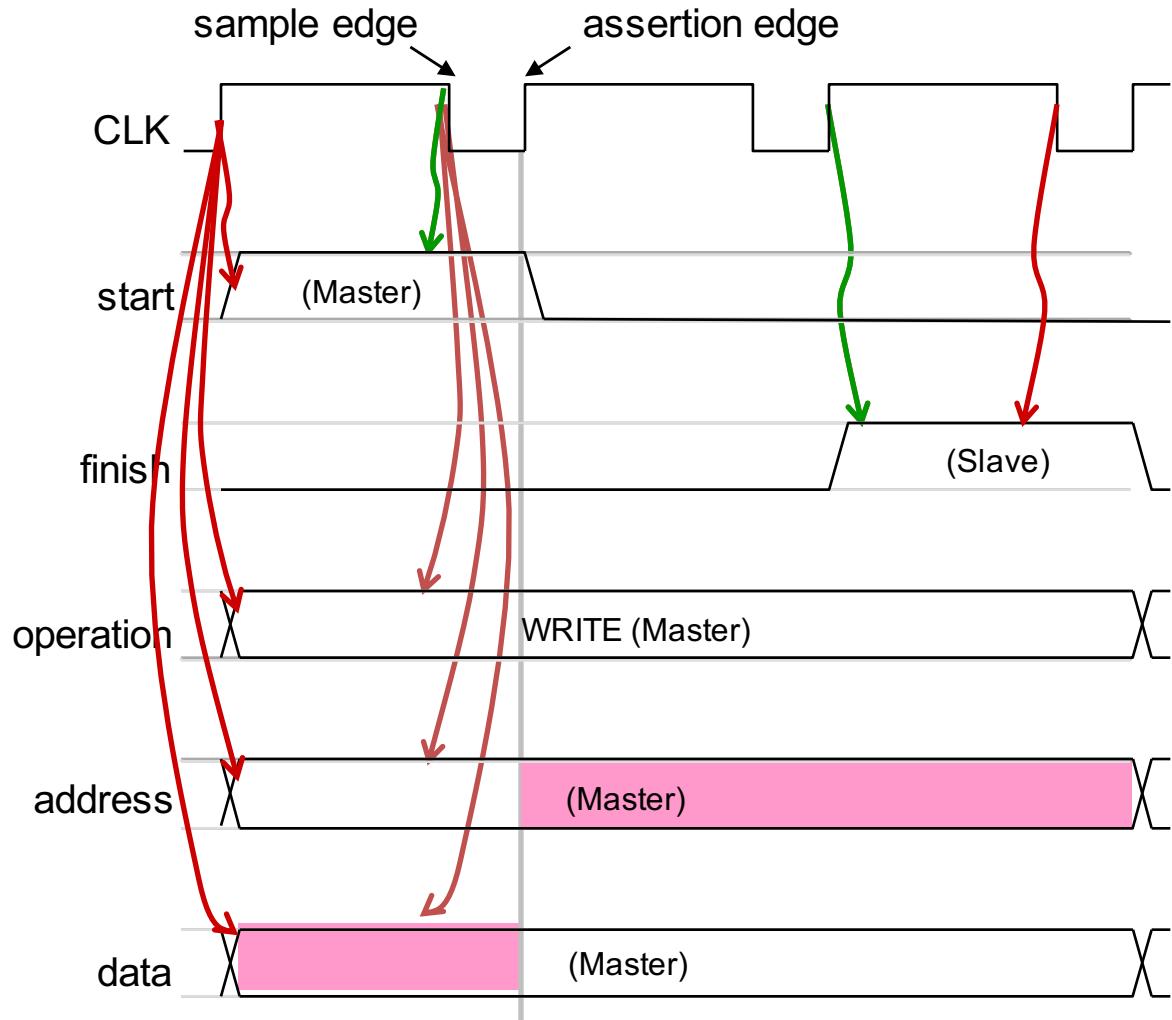


The backplane provides:

- Power
- Common system clock
- Wires for communication



A Parallel Bus Transaction



MASTER:

- 1) Chooses bus operation
- 2) Asserts an address
- 3) Waits for a slave to answer.

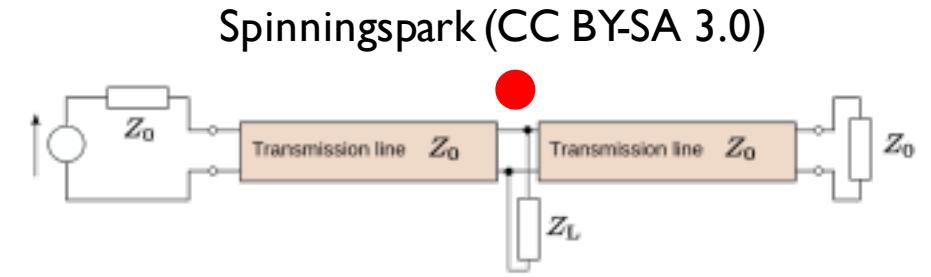
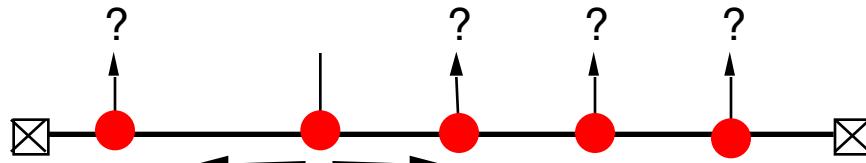
SLAVE:

- 1) Monitors start
- 2) Check address
- 3) If meant for me
 - a) look at bus operation
 - b) do operation
 - c) signal finish of cycle

BUS:

- 1) Monitors start
- 2) Start count down
- 3) If no one answers before counter reaches 0 then "time out"

Bus Lines as Transmission Lines



$$\text{Transmission: } \frac{2Z_L}{Z_0 + 2Z_L}$$

ANALOG ISSUES:

- Propagation times
 - Signals travel at $\sim 18 \text{ cm/ns}$ 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.

$$\text{Reflection: } \frac{-Z_0}{Z_0 + 2Z_L}$$

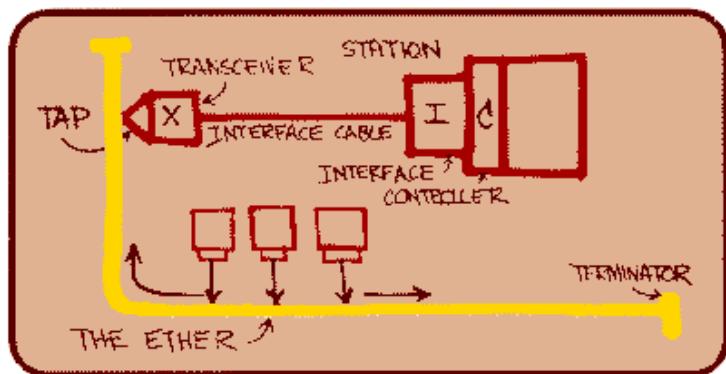
https://en.wikipedia.org/wiki/Reflections_of_signals_on_conducting_lines

Point-to-point Communication

Meanwhile, Outside the Box...

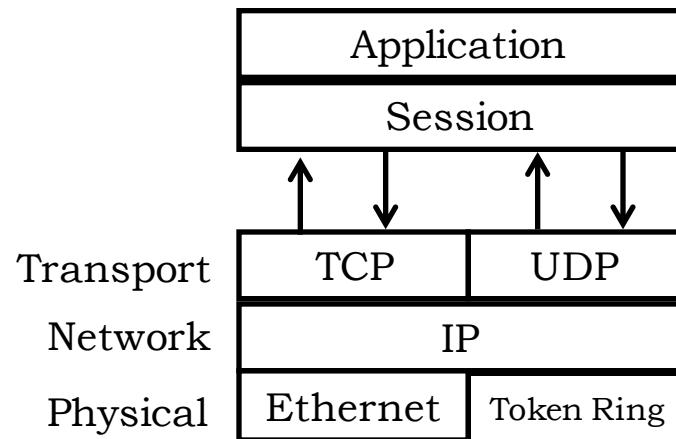
The network as an interface standard.

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-clocked data (no clock, skew!)
 - header (dest), data bits, checksum
- Issues: sharing, contention, arbitration, “backoff”

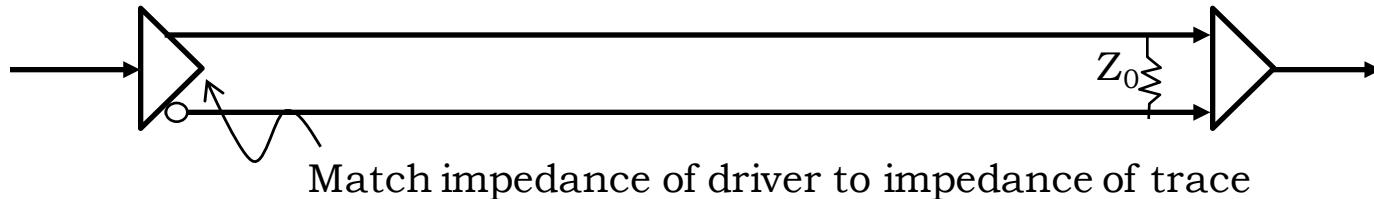
IDEA: Protocol “layers” that isolate application-level interface from low-level physical devices:



Lessons learned: single driver, point-to-point

Differential signaling over controlled impedance trace

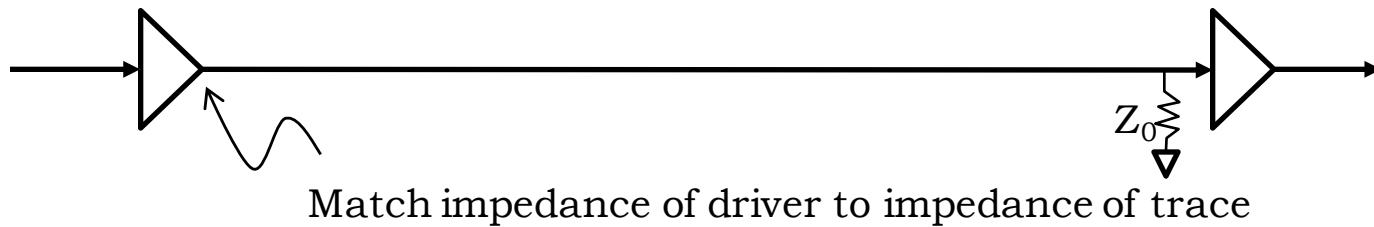
BEST



Match impedance of driver to impedance of trace

Single-ended signaling over controlled impedance trace

OKAY



Match impedance of driver to impedance of trace

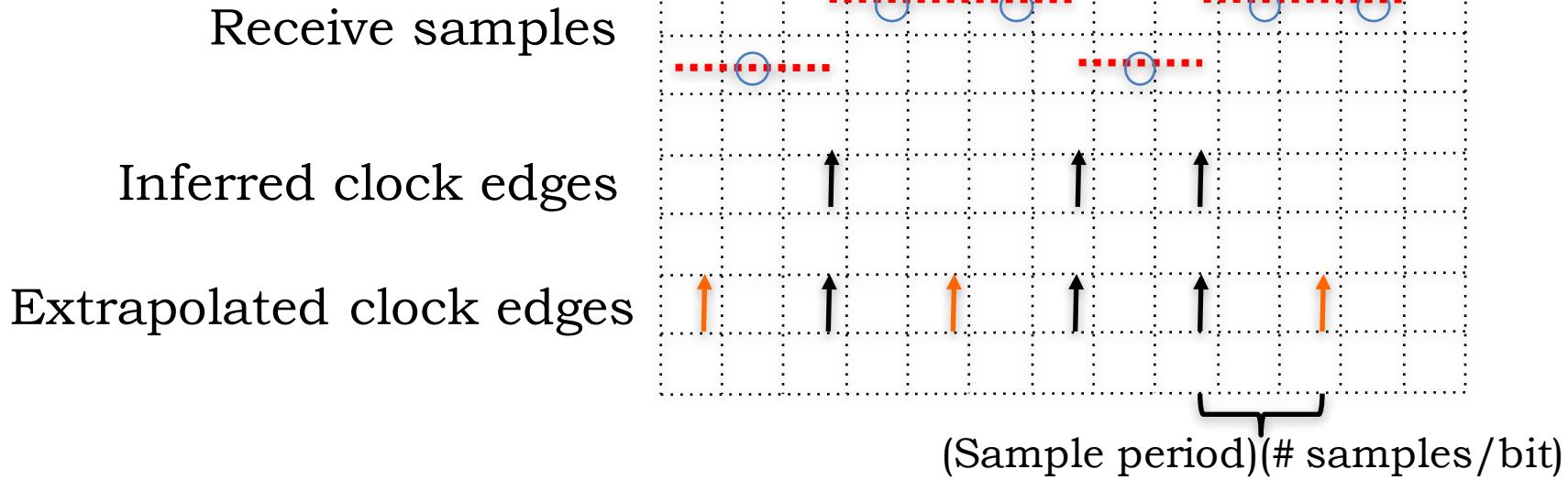
SLOW



Issues:

- Impedance troubles when driving in middle
- Turn-around time when sharing a wire (wired-or glitch)

Lessons learned: clock recovery

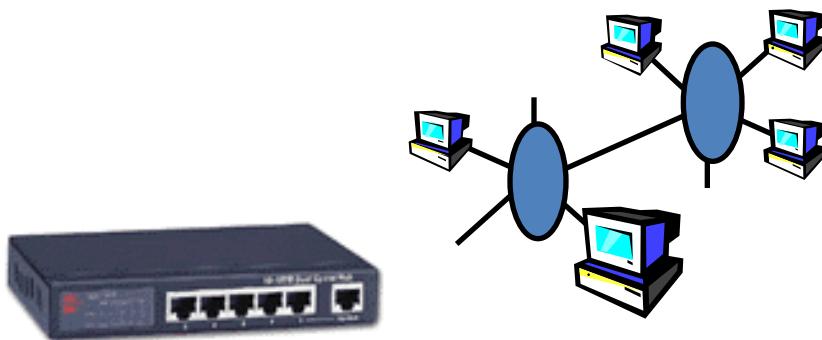
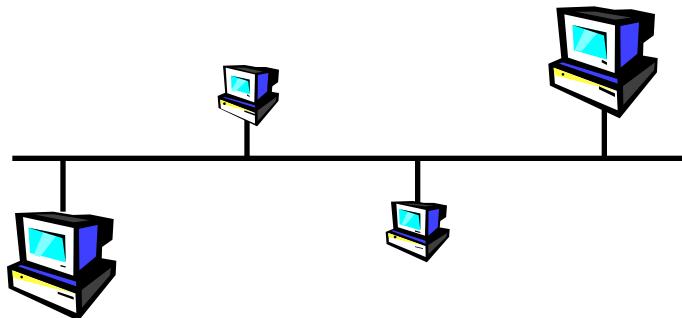


- Receiver can infer presence of clock edge every time there's a transition in the received samples.
- Using sample period, extrapolate remaining edges
 - Now know first and last sample for each bit
 - Choose “middle” sample to determine message bit
- Can't go too long without a clock edge → 8b10b encoding

Serial, Point-to-Point Communications

ETHERNET: Broadcast technology

- Sharing (contention) issues
- Multiple-drop-point issues...
- *bit-serial* (single wire!)
- “Packets” for multi-bit data

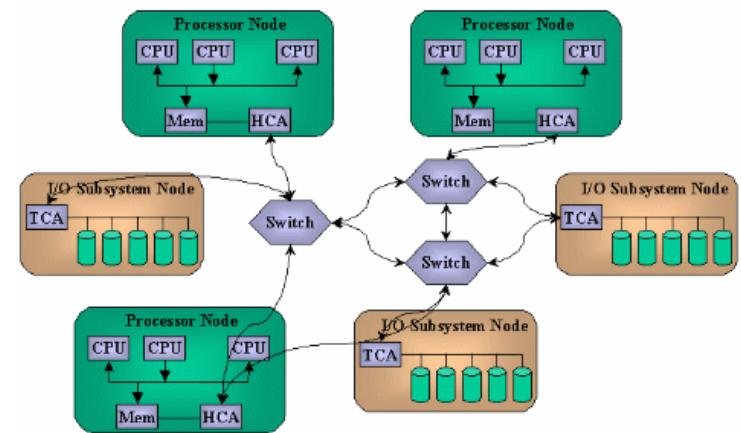


Serial point-to-point bus replacements

- Multi Gbit/sec serial links!
- PCIe, Infiniband, SATA, USB, ...
- Packets, headers
- Switches, routing
- Trend: localized, superfast, serial networks!

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

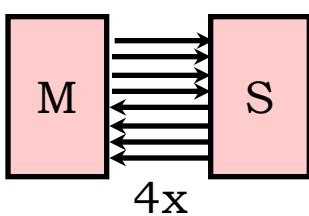
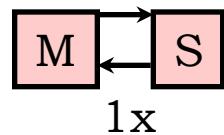
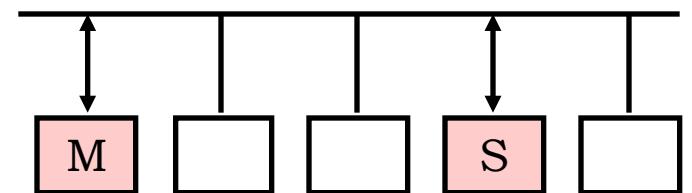


System-level Interconnect

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



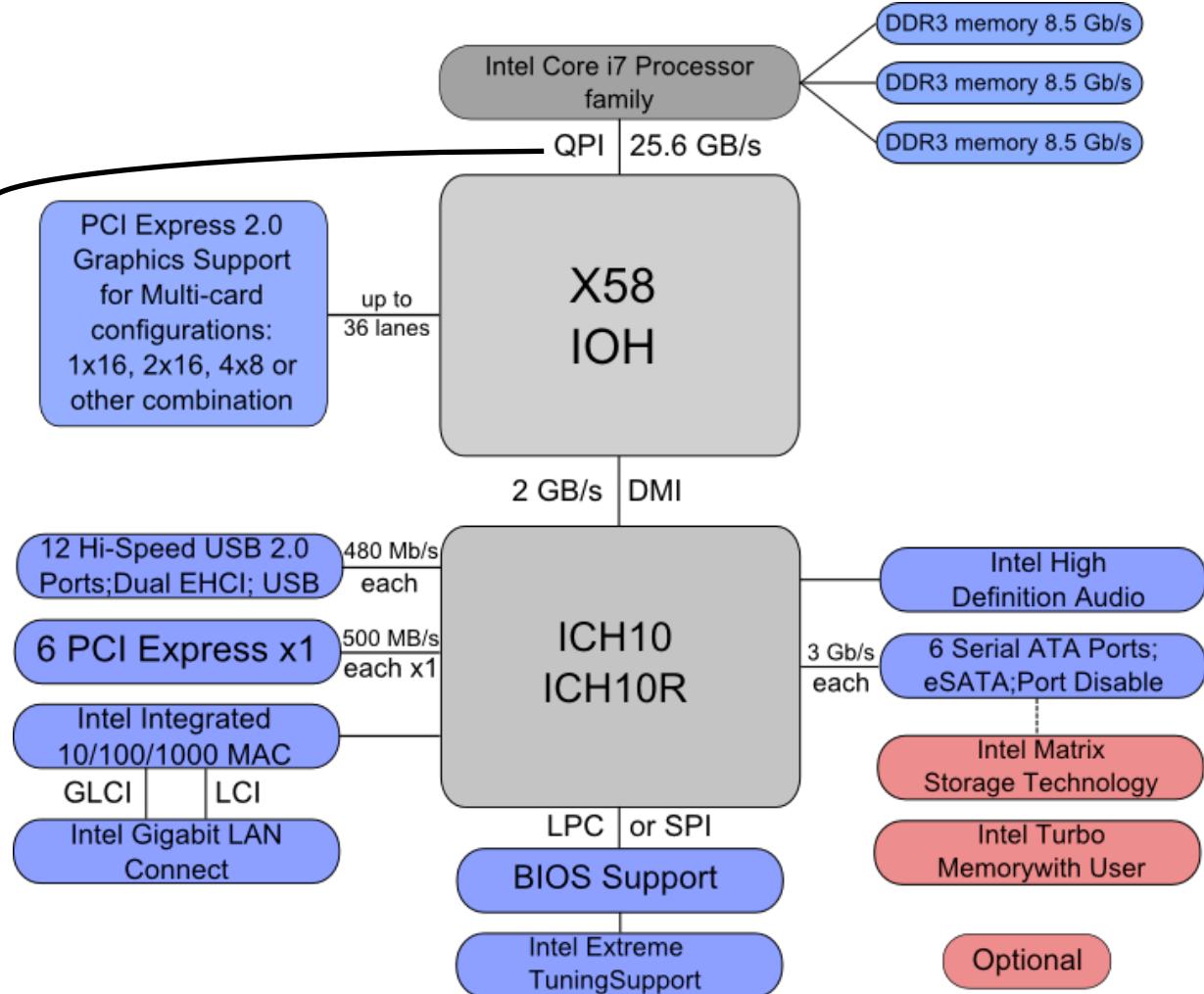
REPLACEMENT: fast unidirectional **serial point-to-point link**

- 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!

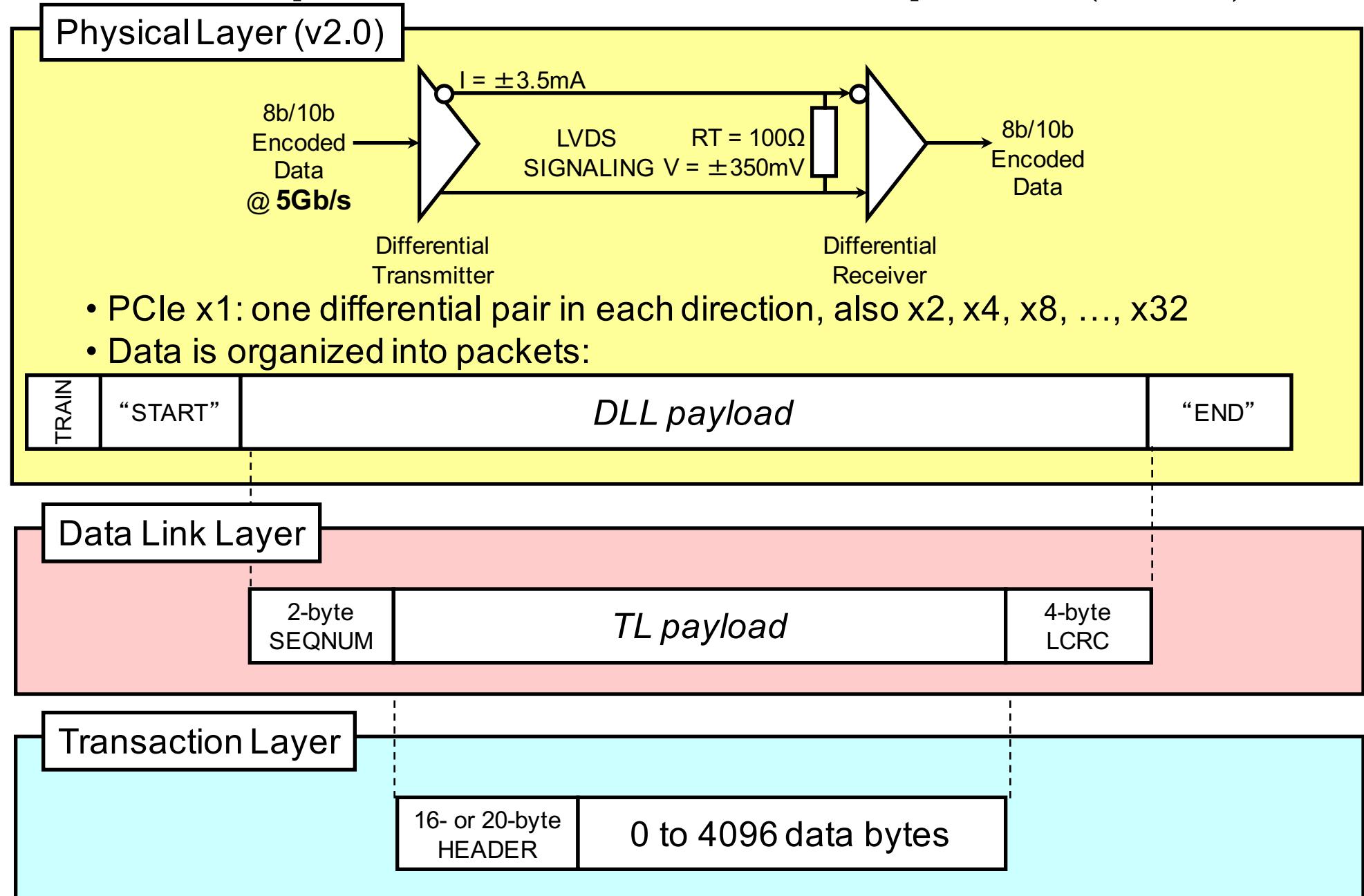
Communications in Today's Computers

QuickPath Interconnect:
→ 20 data + clk
← 20 data + clk
Differential signaling
6.4 GT/s

*One bus to rule them all,
One bus to join them,
One bus to bring them all
And to the CPU bind them.*



Example serial link: PCI Express (PCIe)



Communication Topologies

Communication Topologies

asymptotic cost/performance tradeoffs

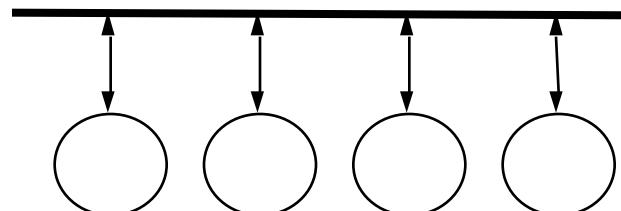
Goal: enable communications between n components

- Each point-to-point link requires one hardware unit.
- Each point-to-point communication requires one time unit.
- Each link operates independently

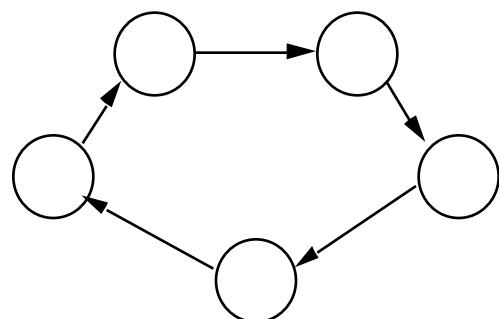
1-dimensional approaches:

BUS

Shared communication channel allows only one message at a time



Throughput	$O(1)$
Latency	$O(1)$
Cost	$O(n)$

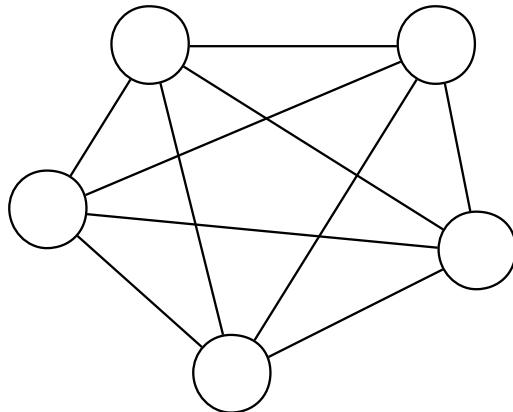


RING

Each component has link to next component on ring

Throughput	$O(n)$
Latency	$O(n)$
Cost	$O(n)$

Quadratic-cost Topologies



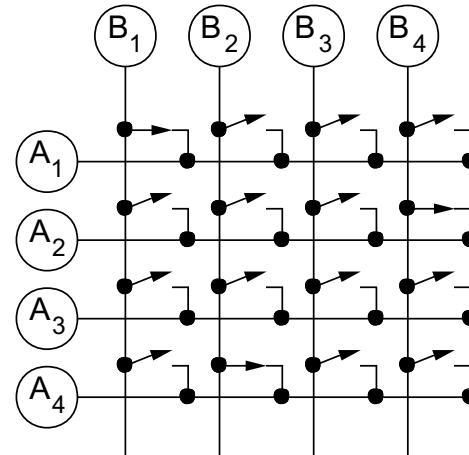
COMPLETE GRAPH

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

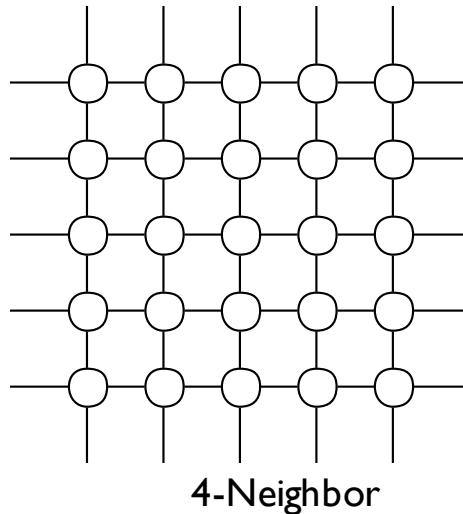
- Switch dedicated between each pair of nodes
- Each A_i can be connected to one B_j at any time
- Special cases:
 - A = processors, B = memories
 - A, B same type of node
 - A, B same nodes (complete graph)



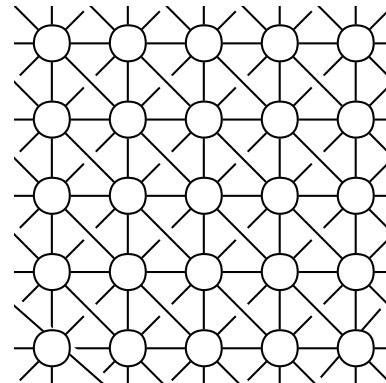
Throughput	$O(n)$
Latency	$O(1)$
Cost	$O(n^2)$

Mesh Topologies

2-Dimensional Meshes



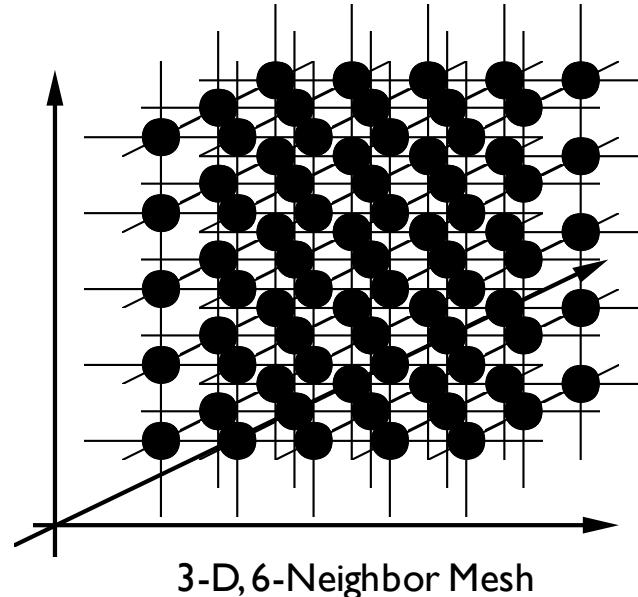
4-Neighbor



8-Neighbor

Throughput	$O(n)$
Latency	$O(\sqrt{n})$
Cost	$O(n)$

Nearest-neighbor connectivity:
Point-to-point interconnect
- minimizes delays
- minimizes “analog” effects
Store-and-forward
(some overhead associated with communication routing)

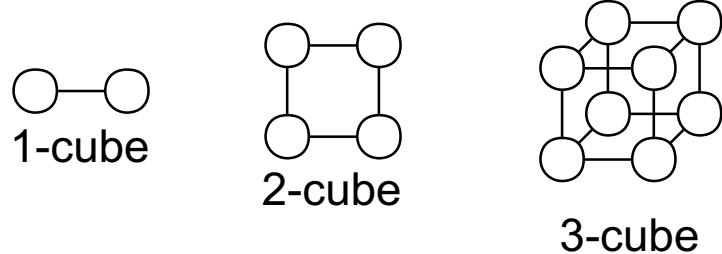


Throughput	$O(n)$
Latency	$O(\sqrt[3]{n})$
Cost	$O(n)$

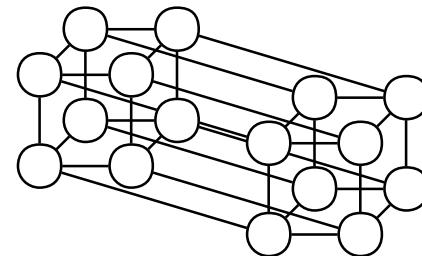
Logarithmic-latency Networks

HYPERCUBE

D dimensions $\rightarrow 2^D$ nodes
Each node has D links

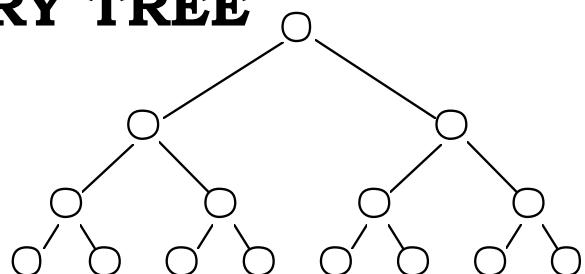


Throughput	$O(n \log_D n)$
Latency	$O(\log_D n)$
Cost	$O(n \log_D n)$



4-cube

BINARY TREE



Throughput	$O(n)$
Latency	$O(\log_2 n)$
Cost	$O(n)$

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	$O(n^2)$	$O(1)$	$O(n)$
ID 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)

Communications Futures

Backplane buses have evolved into point-to-point links

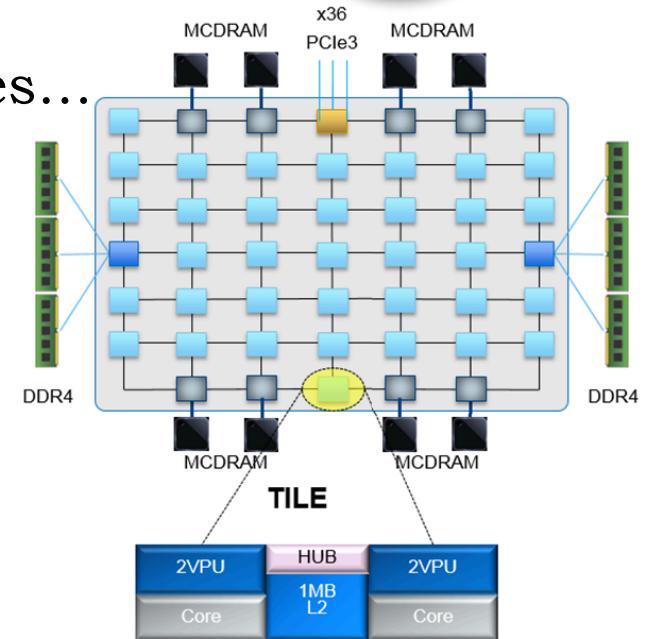
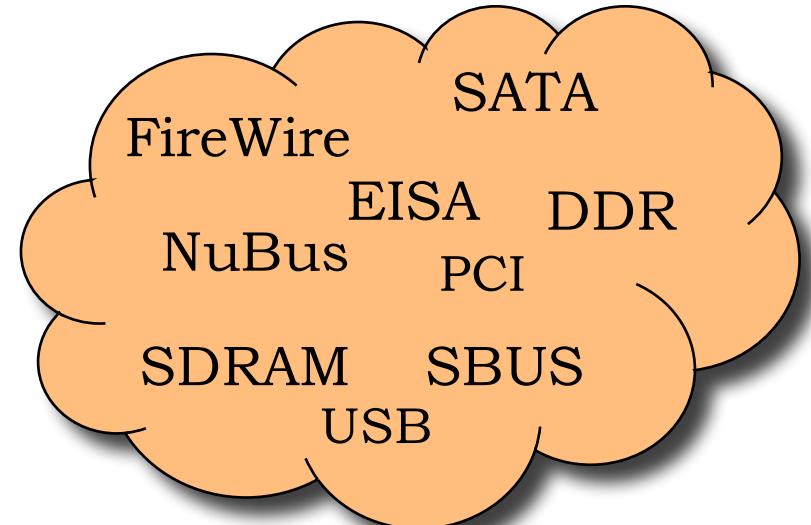
- + links operate independently
- + links can be managed in groups
- + packetized data deals with errors

Specialized buses for memory

Networked “peripherals” for mobile devices...

New-generation communications...

- how should 100 (1000?) cores communicate?



21. Parallel Processing

6.004x Computation Structures
Part 3 – Computer Organization

Copyright © 2016 MIT EECS

Instruction-level Parallelism

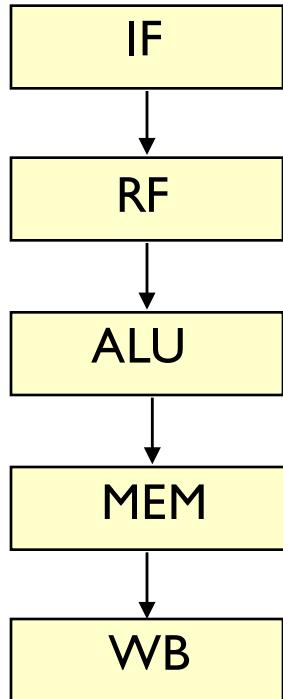
Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \frac{\text{Cycles}}{\text{Instruction}} \frac{\text{Time}}{\text{Cycle}} \\ \qquad \qquad \qquad \qquad \qquad \text{CPI} \qquad t_{\text{CLK}}$$

- 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

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

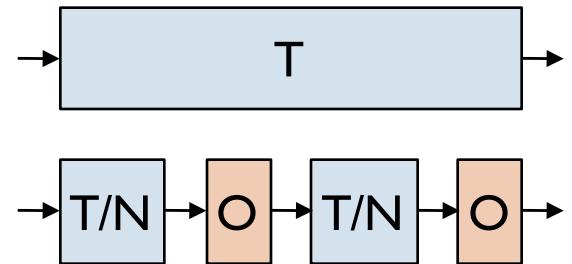


Improving 5-stage Pipeline Performance

- Lower t_{CLK} : **deeper pipelines**
 - Overlap more instructions

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



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)**

Instruction Level Parallelism (ILP)

Sequential Code

loop:

```
LD(n, r1)
CMPLT(r31, r1, r2)
BF(r2, done)
LD(r, r2)
LD(n, r1)
MUL(r1, r2, r3)
ST(r3, r)
LD(n, r4)
SUBC(r4, 1, r4)
ST(r4, n)
BR(loop)
```

done:

$$r = \prod_{i=1}^n i$$

“Safe” Parallel Code

loop:

```
LD(n, r1)
CMPLT(r31, r1, r2)
BF(r2, done)
LD(r, r2) LD(n, r1) LD(n, r4)
MUL(r1, r2, r3) SUBC(r4, 1, r4)
ST(r3, r) ST(r4, n) BR(loop)
```

done:

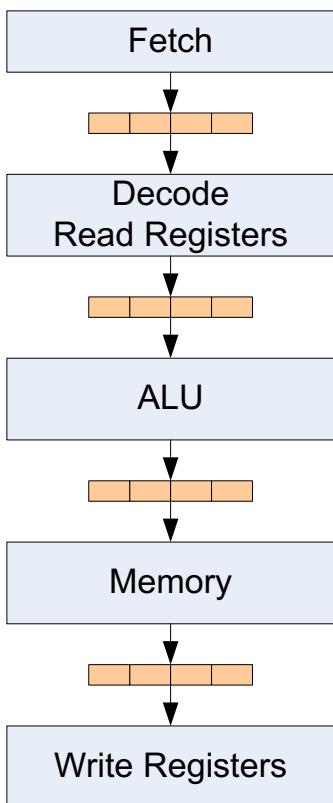
- Read-after-write
- Write-after-write
- Write-after-read



These last two can be solved with renaming, i.e., giving each result a unique register name.

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?



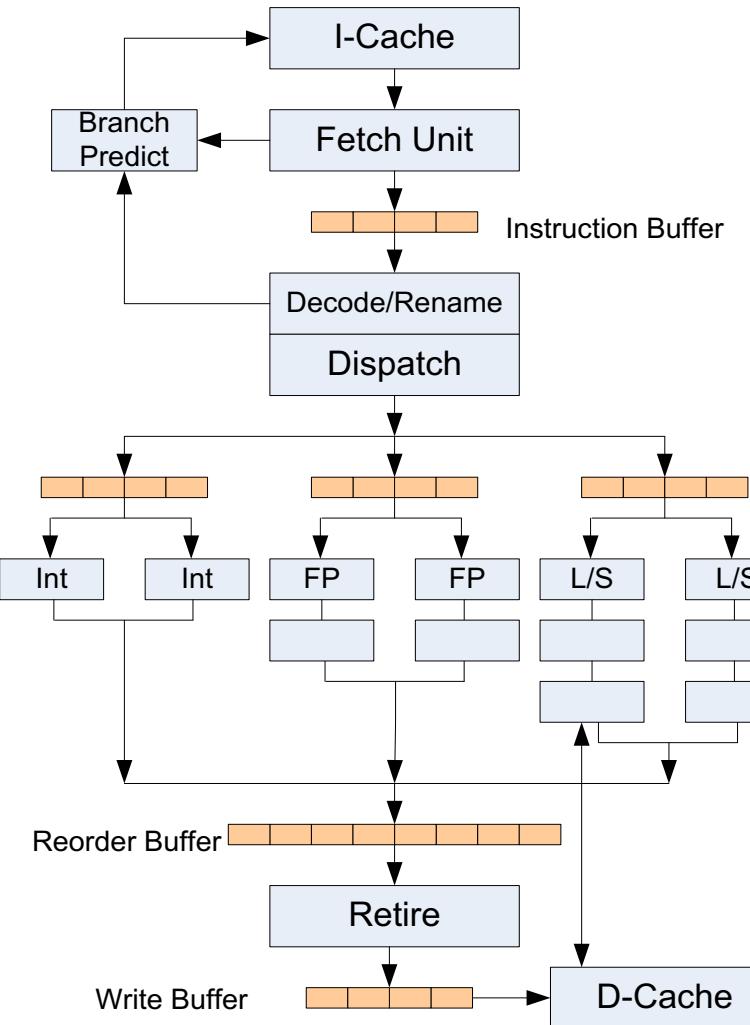
See <http://people.ee.duke.edu/~sorin/ece252/lectures/3-superscalar.pdf>

A Modern Out-of-Order Superscalar Processor

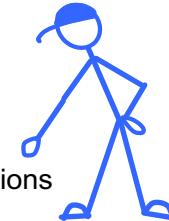
Needed to
avoid high
 CPI_{STALL} on
deep pipelines



In Order ↑
↓ Out Of Order
↓ In Order



For OoO:
determine when
operands are
ready for inst.



Make sure side-
effects happen
in correct
order!

Limits To Single-Processor Performance

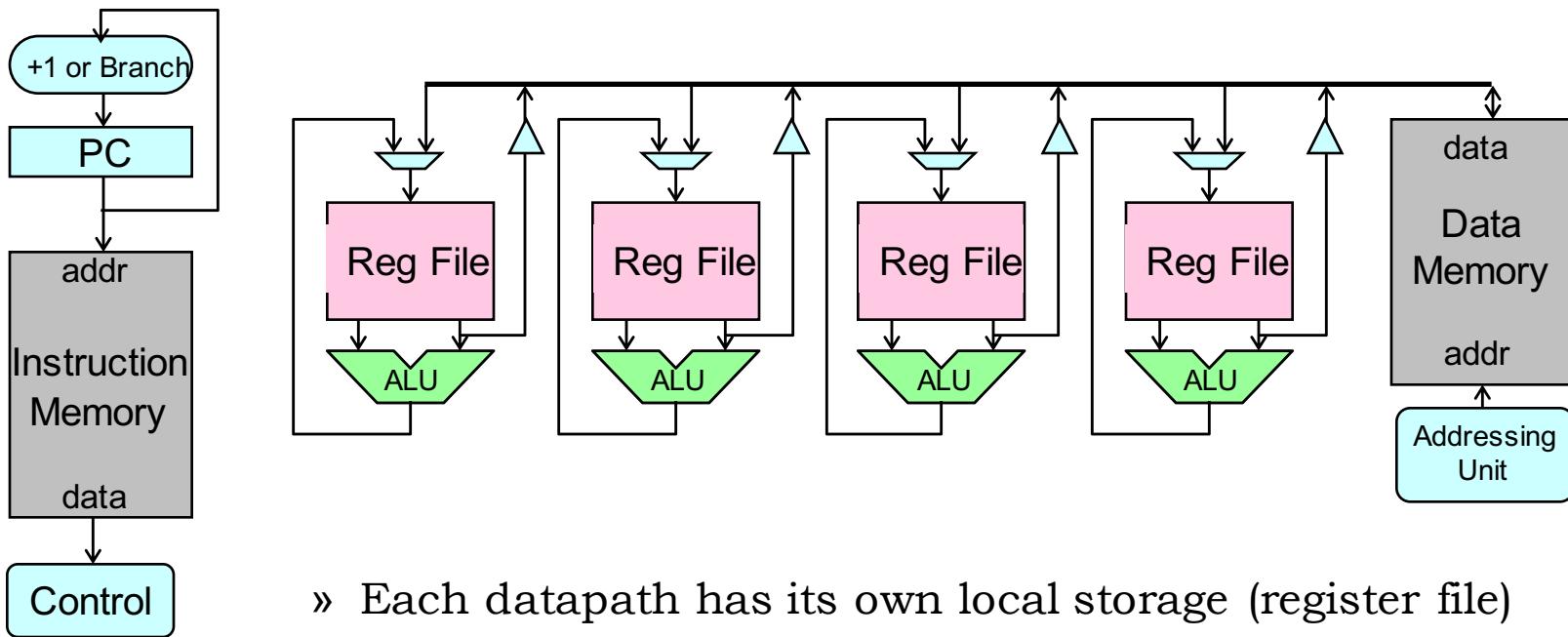
- Pipeline depth: getting close to pipelining limits
 - Clocking overheads, CPI degradation
- Branch prediction & memory latency limit the practical benefits of out-of-order execution
- Power grows superlinearly with higher frequency & more OoO logic
- Extreme design complexity
- Limited ILP → Must exploit DLP and TLP
 - Data-Level Parallelism: Vector extensions, GPUs
 - Thread-Level Parallelism: Multiple threads and cores

Data-level Parallelism

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

Vector Code Example

```
for (i = 0; i < 16; i++) x[i] = a[i] + b[i];
```

Beta assembly

```
CMOVE(16, R0)  
loop: LD(R1, 0, R4)  
      LD(R2, 0, R5)  
      ADDC(R1, 4, R1)  
      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 = $1 + 10*15 + 9 = 160$

Equivalent vector assembly

```
LD.V(R1, 0, V1)  
LD.V(R2, 0, V2)  
ADD.V(V1, V2, V3)  
ST.V(V3, 0, R3)
```

of cycles = 4

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 istrue or isfalse.
ADDC.V.iftrue(V3, 3, V3)
```

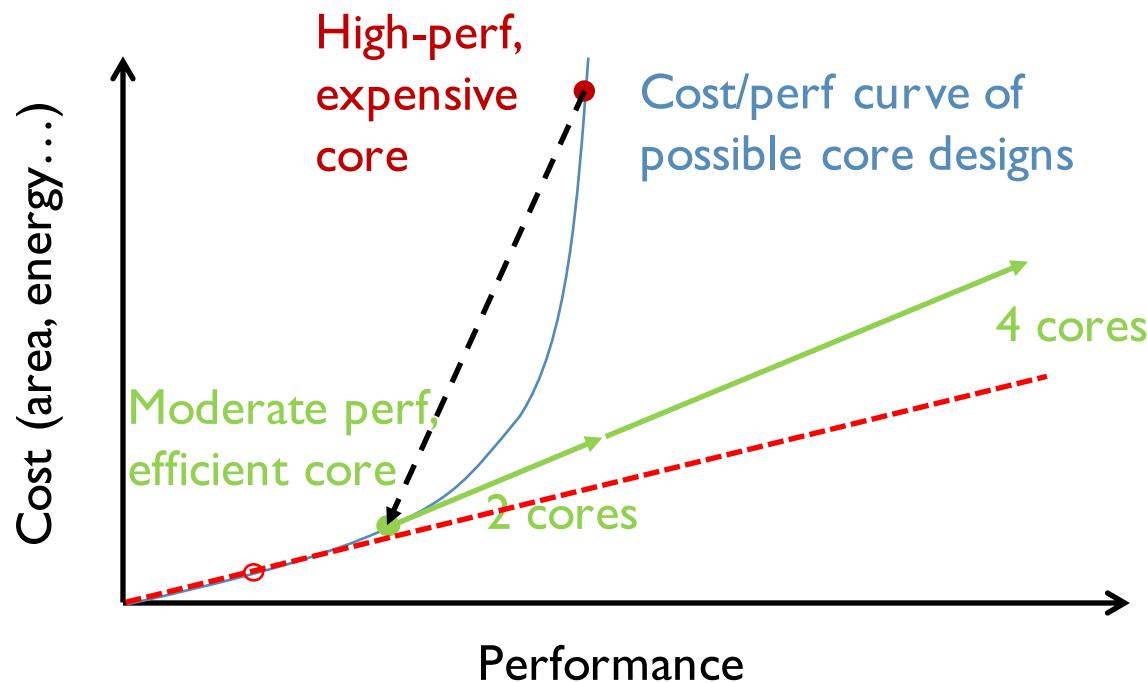
Vector Processing Implementations

- 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)

Thread-level Parallelism

Multicore Processors

If applications have a lot of parallelism, using a larger number of simpler cores is more efficient!



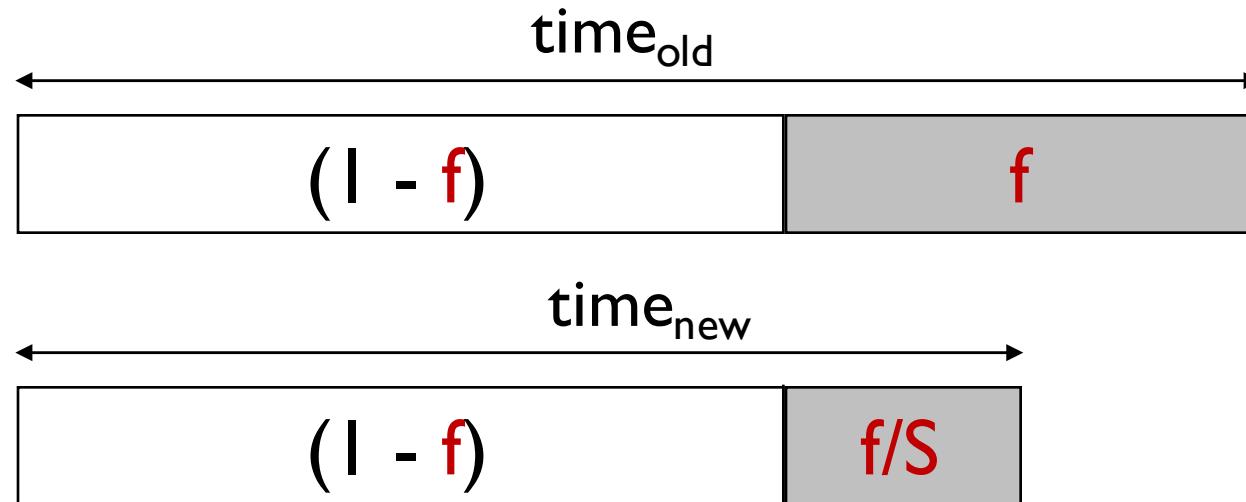
What is the optimal tradeoff between core cost and number of cores?

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

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

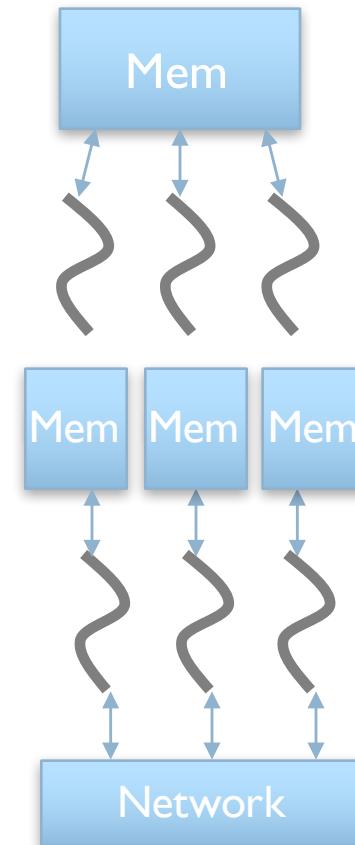
$$f = 0.9, S=\infty \rightarrow S_{\text{overall}} = 10$$

What f do you need to use a 1000-core machine well?

$$S_{\text{overall}} = 500 \rightarrow f = 0.998$$

Thread-Level Parallelism

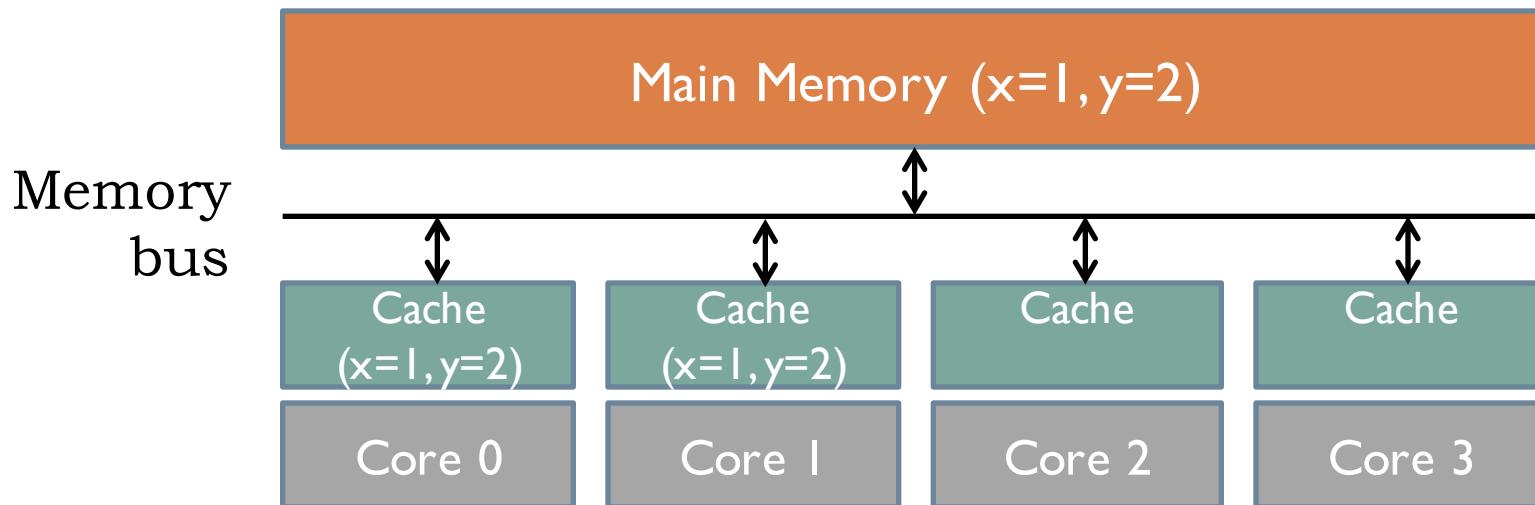
- Divide computation among multiple threads of execution
 - Each thread executes a different instruction stream
 - More flexible than vector processing, but more expensive
- Communication models:
 - Shared memory:
 - Single address space
 - Implicit communication by memory loads & stores
 - Message passing:
 - Separate address spaces
 - Explicit communication by sending and receiving messages



Shared Memory & Caches

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);
```

What Are the Possible Outcomes?

Thread A

```
x = 3;  
print(y);
```

$\$_1: x = \text{X } 3$
y = 2

Thread B

```
y = 4;  
print(x);
```

$\$_2: x = 1$
y = $\text{X } 4$

Plausible execution sequences:

SEQUENCE

x=3; print(y); y=4; print(x);
x=3; y=4; print(y); print(x);
x=3; y=4; print(x); print(y);
y=4; x=3; print(x); print(y);
y=4; x=3; print(y); print(x);
y=4; print(x); x=3; print(y);

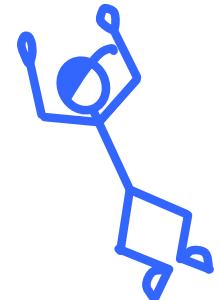
A prints

2
2
2
2
2
2

B prints

1
1
1
1
1
1

Hey, we get the same answer every time... Let's go build it!



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);
```

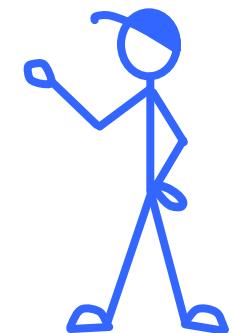
Plausible Uniprocessor execution sequences:

SEQUENCE

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

A prints B prints

*Notice that
the outcome
2, 1 does not
appear in
this list!*



Sequential Consistency

Semantic constraint:

Result of executing N parallel threads should correspond to *some* interleaved execution on a single processor.

Shared Memory

```
int x=1, y=2;
```

Thread A

```
x = 3;  
print(y);
```

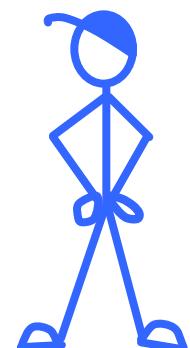
Thread B

```
y = 4;  
print(x);
```

Possible printed values: 2, 3; 4, 3; 4, 1.
(each corresponds to at least one interleaved execution)

IMPOSSIBLE printed values: 2, 1
(corresponds to NO valid interleaved execution).

Were'n't
caches
supposed to
be invisible
to programs?



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:

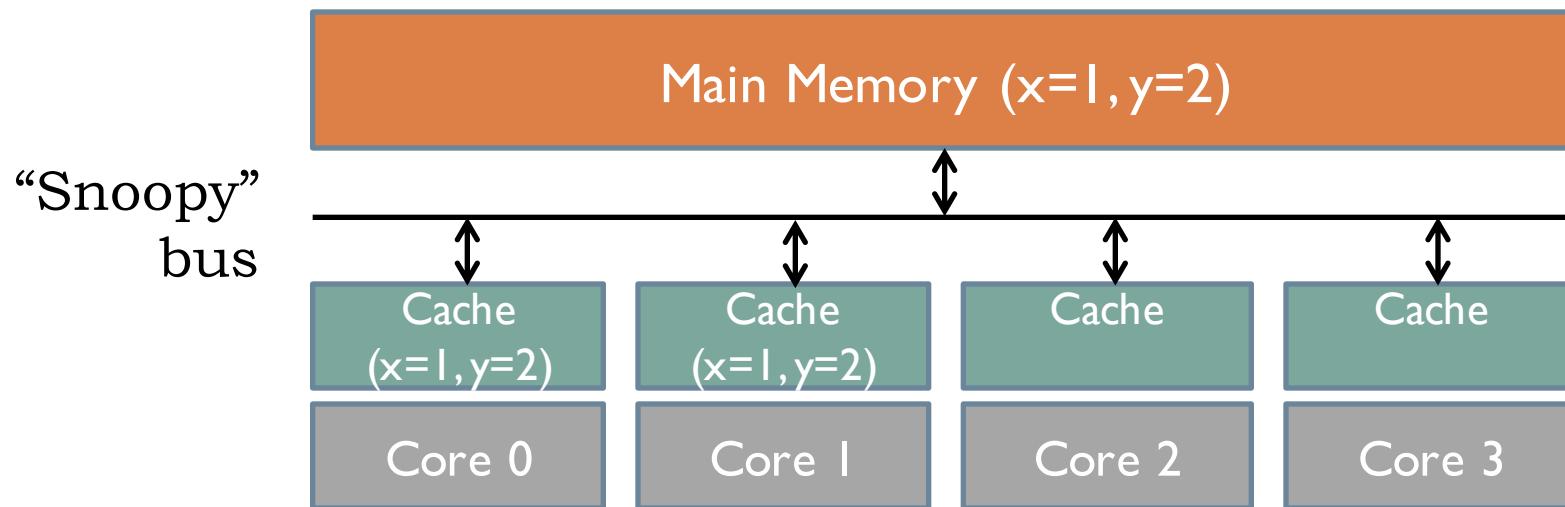
- Weak consistency, by default;
- MEMORY BARRIER instruction: stalls thread until all previous memory operations have completed.

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.

Cache Coherence

Fix: “Snoopy” Cache Coherence Protocol

Idea: Have caches communicate over shared bus, letting other caches know when a shared cached value changes



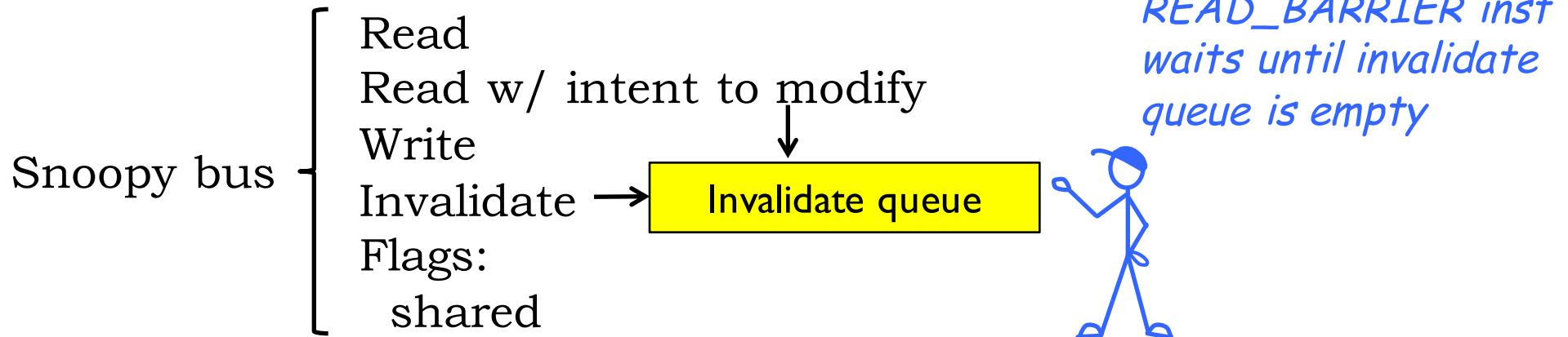
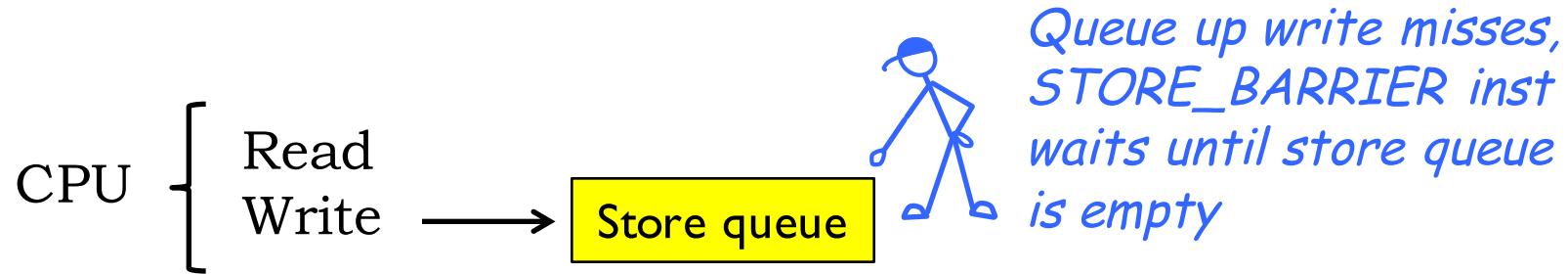
Goal: minimize contention for snoopy bus by communicating only when necessary, i.e., when there's a shared value.

Example: MESI Cache Coherence Protocol

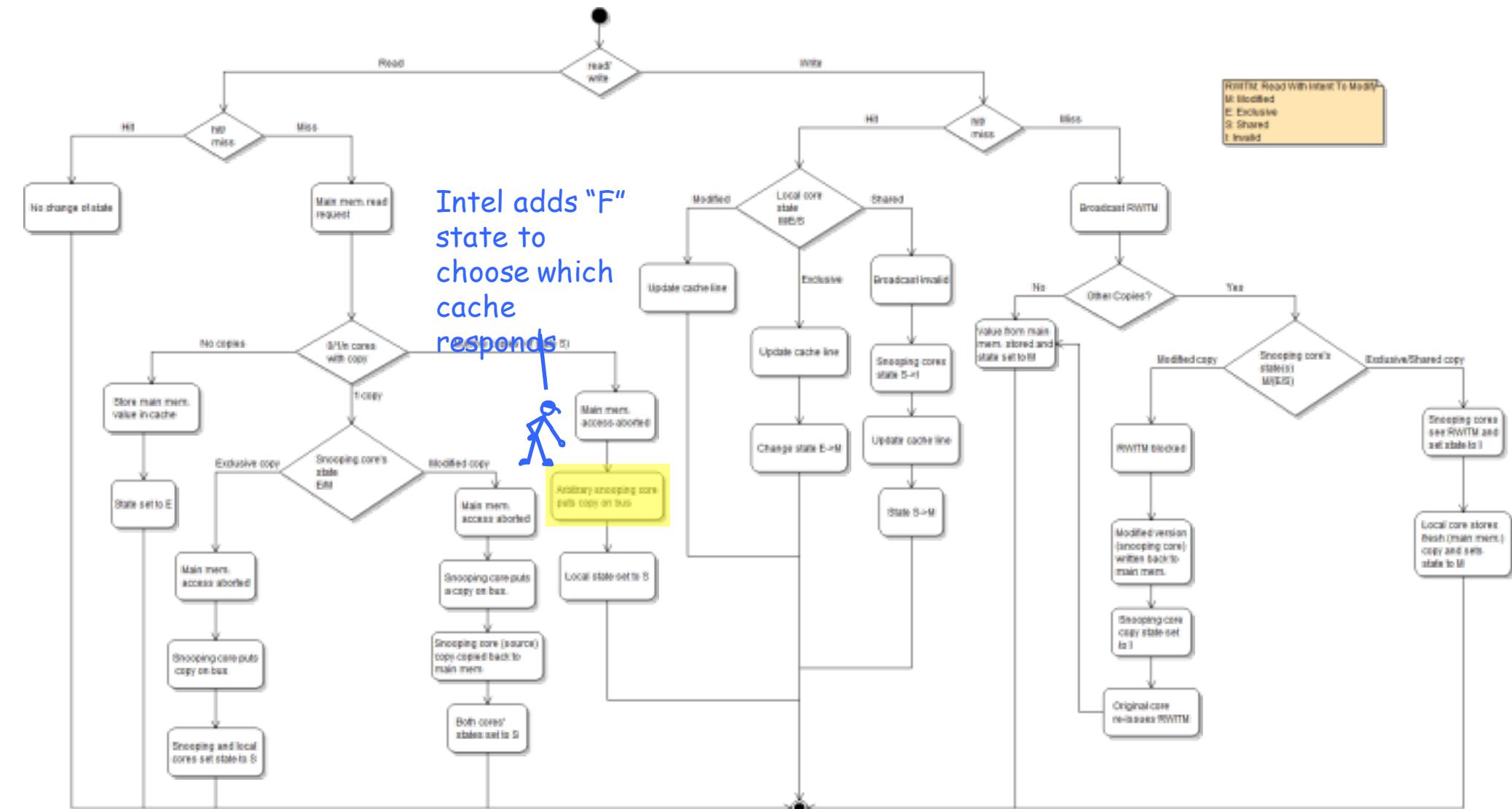
Cache line:	State	Tag	Data
-------------	-------	-----	------

- **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).

The Cache Has Two Customers!



MESI Activity Diagram



CC0: https://en.wikipedia.org/wiki/MESI_protocol#/media/File:MESI_protocol_activity_diagram.png

Cache Coherence in Action

Thread A

- 1 $x = 3;$
- 4 $\text{print}(y);$

$\$_0: [S] x = 1, [S] y = 2$

Thread B

- 2 $y = 4;$
- 3 $\text{print}(x);$

$\$_1: [S] x = 1, [S] y = 2$

1. $x = 3 \rightarrow \$_0$ sends invalidate x, update cache

$\$_0: [M] x = 3, [S] y = 2$

$\$_1: [S] y = 2$

2. $y = 4 \rightarrow \$_1$ sends invalidate y, update cache

$\$_0: [M] x = 3$

$\$_1: [M] y = 4$

3. $\text{print}(x) \rightarrow \$_1$ read x, $\$_0$ responds with Shared flag, update mem

$\$_0: [S] x = 3$

$\$_1: [S] x = 3, [M] y = 4$

4. $\text{print}(y) \rightarrow \$_0$ read y, $\$_1$ responds with Shared flag, update mem

$\$_0: [S] x = 3, [S] y = 4$

$\$_1: [S] x = 3, [S] y = 4$

Parallel Processing Summary

Prospects for future CPU architectures:

Pipelining - Well understood, but mined-out

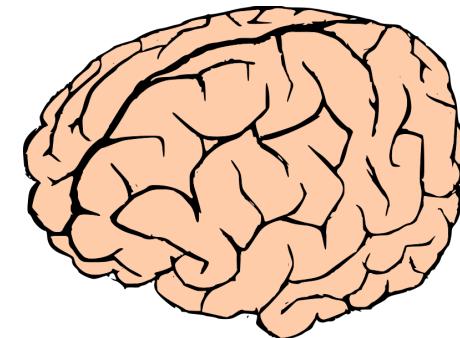
Superscalar - At its practical limits

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!

6.004 Wrap-up

From Atoms to Amazon

6.004

Parallelism & communication

Cloud

Virtual Memory

Virtual Machines

Operating System

Programming languages

Interpretation & Compilation

Instruction set + memory

Data and Control structures

Programmable architectures

Bits, Logic gates

FSMs + Datapaths

Lumped component model

Digital Circuits

*Insulator, conductor,
semiconductor*

Devices

Materials

Atoms

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 \Rightarrow 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*

6.004: The Big Lesson

You've built, debugged, understood a complex computer from FETs to OS... what have you learned?

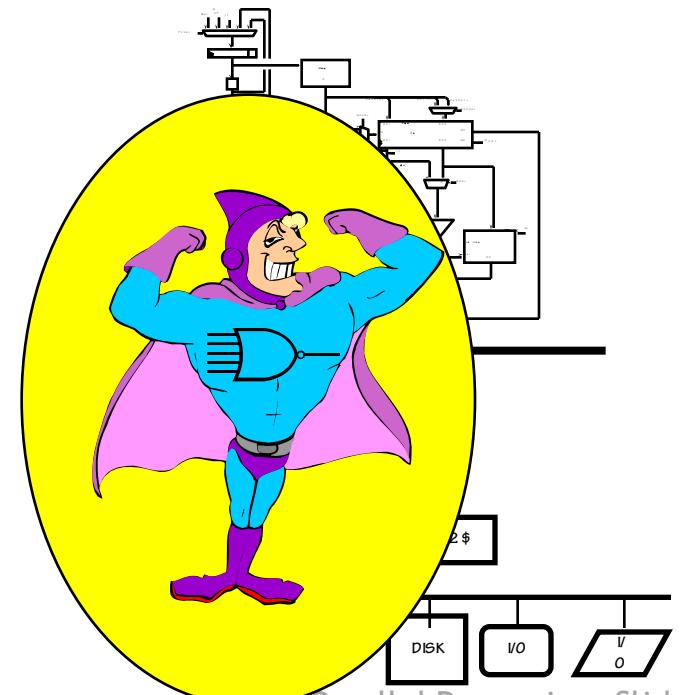
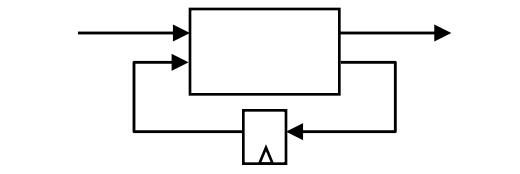
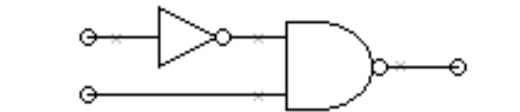
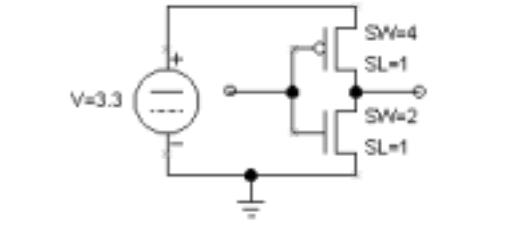
Engineering Abstractions:

- 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!



Things to look forward to...

6.004 is only an appetizer!

Processors
Superscalars
Deep pipelines
Multicores

Languages & Models
Python/Java/Ruby/...
Objects/Streams/Aspects
Networking

Tools
Design Languages
FPGA prototyping
Timing Analyzers

Algorithms
Arithmetic
Signal Processing
Language
implementation

Systems Software
Storage
Virtual Machines
Networking

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



MOSFET
transistors

Von Neumann
Architectures

Synchronous
Clocked
Systems

Boolean
Logic

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

