CS152
Computer Architecture and Engineering
Lecture 3

Performance, Technology & Delay Modeling

Jan 27, 1999

John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

Outline of Today's Lecture

- **Review : Finish ISA/MIPS details (10 minutes)**
- **Performance and Technology (15 minutes)**
- **Administrative Matters and Questions (2 minutes)**
- **Delay Modeling and Gate Characterization (20 minutes)**
- **Questions and Break (5 minutes)**
- **Clocking Methodologies and Timing Considerations (25 minutes)**

Summary: Instruction set design (MIPS)

- **Use general purpose registers with a load-store architecture: YES**
- **Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPR & 32 FPR**
- **Support basic addressing modes: *displacement* (with address offset of 12 to 16 bits), *immediate* (size 8 to 16 bits), and *register deferred*: YES: 16 bit immediate, displacement (disp=0 ⇒ register deferred)**
- **All addressing modes apply to all data transfer instructions : YES**
- **Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : Fixed**
- **Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES**
- **Support most common instructions, since they will dominate:**
  load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch* (with a PC-relative address at least 8-bits long), *jump, call, and return:
  YES, 16b relative address
- **Aim for a minimalist instruction set: YES**

Summary: Salient features of MIPS I

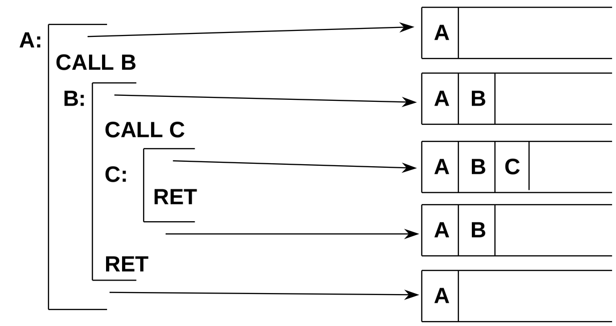
- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (+ HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - no indirection, scaled
- **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =,≠
  - no integer condition codes
- **Support for 8bit, 16bit, and 32bit integers**
- **Support for 32bit and 64bit floating point.**

Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Branch/jump and link put the return addr. PC+4 into the link register (R31)
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates ops are zero extended to 32 bits
  - arithmetic immediates ops are sign extended to 32 bits (including addu)
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
  - it cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



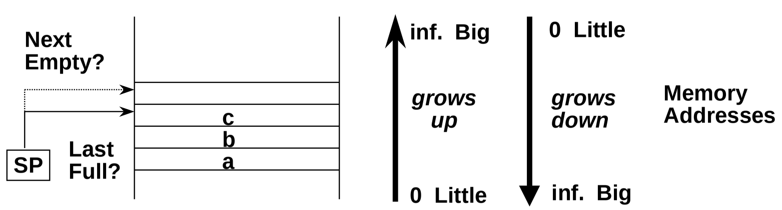
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

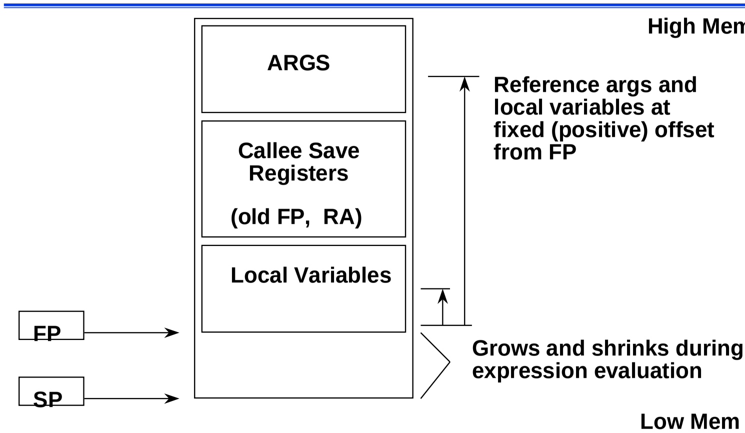
Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?
Consider case of stack *growing down* (MIPS)

Last Full		Last Empty	
POP:	Read from Mem(SP) Increment SP	POP:	Increment SP Read from Mem(SP)
PUSH:	Decrement SP Write to Mem(SP)	PUSH:	Write to Mem(SP) Decrement SP

Call-Return Linkage: Stack Frames



- Many variations on stacks possible (up/down, last pushed / next)
- Compilers normally keep scalar variables in registers, not memory!

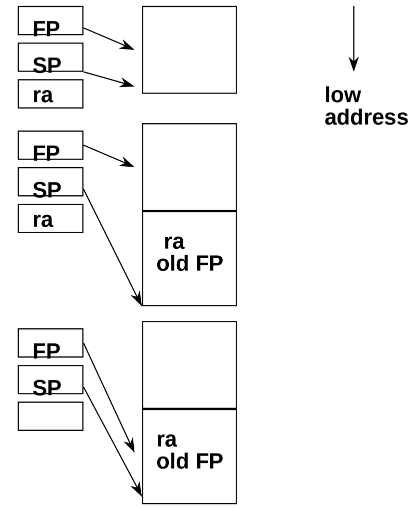
MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	

16	s0	callee saves
. . .		(callee must save)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

MIPS / GCC Calling Conventions

```
fact:
  addiu   $sp, $sp, -32
  sw      $ra, 20($sp)
  sw      $fp, 16($sp)
  addiu   $fp, $sp, 32
. . .
  sw      $a0, 0($fp)
...
  lw      $31, 20($sp)
  lw      $fp, 16($sp)
  addiu   $sp, $sp, 32
  jr      $31
```



First four arguments passed in registers.

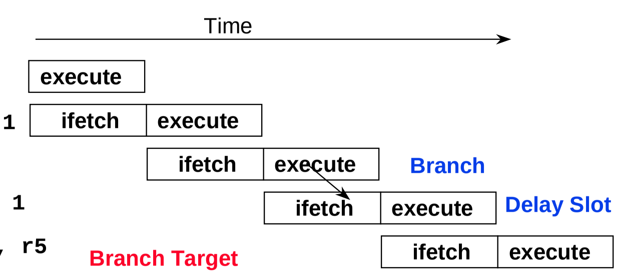
Delayed Branches

```
      li    r3, #7
      sub   r4, r4, 1
      bz    r4, LL
      addi  r5, r3, 1
      subi  r6, r6, 2
LL:   slt   r1, r3, r5
```

- In the "Raw" MIPS, the instruction after the branch is executed even when the branch is taken?
  - This is hidden by the assembler for the MIPS "virtual machine"
  - allows the compiler to better utilize the instruction pipeline (???)

Branch & Pipelines

```
    li r3, #7
    sub r4, r4, 1
    bz r4, LL
    addi r5, r3, 1
LL: slt  r1, r3, r5
```



By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?

Is this a violation of the ISA abstraction?

Performance

- Purchasing perspective
  - given a collection of machines, which has the
    - best performance ?
    - least cost ?
    - best performance / cost ?
- Design perspective
  - faced with design options, which has the
    - best performance improvement ?
    - least cost ?
    - best performance / cost ?
- Both require
  - basis for comparison
  - metric for evaluation
- Our goal is to understand cost & performance implications of architectural choices

Two notions of "performance"

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concorde	3 hours	1350 mph	132	178,200

Which has higher performance?

- Time to do the task (Execution Time)
  – execution time, response time, latency
- Tasks per day, hour, week, sec, ns. .. (Performance)
  – throughput, bandwidth

Response time and throughput often are in opposition

Definitions

- Performance is in units of things-per-second
  - bigger is better
- If we are primarily concerned with response time
  - $\text{performance}(x) = \dfrac{1}{\text{execution\_time}(x)}$

" X is n times faster than Y" means

$$n = \frac{\text{Performance(X)}}{\text{Performance(Y)}}$$

Example

- Time of Concorde vs. Boeing 747?
  - Concord is 1350 mph / 610 mph = 2.2 times faster = 6.5 hours / 3 hours
- Throughput of Concorde vs. Boeing 747 ?
  - Concord is 178,200 pmph / 286,700 pmph = 0.62 "times faster"
  - Boeing is 286,700 pmph / 178,200 pmph = 1.60 "times faster"
- Boeing is 1.6 times ("60%") faster in terms of throughput
- Concord is 2.2 times ("120%") faster in terms of flying time

We will focus primarily on execution time for a single job

Lots of instructions in a program => Instruction throughput important!

Basis of Evaluation

Pros		Cons
• representative	Actual Target Workload	• very specific • non-portable • difficult to run, or measure • hard to identify cause
• portable • widely used • improvements useful in reality	Full Application Benchmarks	•less representative
• easy to run, early in design cycle	Small "Kernel" Benchmarks	• easy to "fool"
• identify peak capability and potential bottlenecks	Microbenchmarks	• "peak" may be a long way from application performance

SPEC95

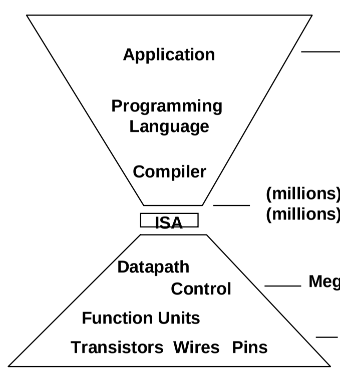
° Eighteen application benchmarks (with inputs) reflecting a technical computing workload

° Eight integer
  - go, m88ksim, gcc, compress, li, ijpeg, perl, vortex

° Ten floating-point intensive
  - tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fppp, wave5

° Must run with standard compiler flags
  - eliminate special undocumented incantations that may not even generate working code for real programs

Metrics of performance



Answers per month
Useful Operations per second

(millions) of Instructions per second – MIPS
(millions) of (F.P.) operations per second – MFLOP/s

Megabytes per second

Cycles per second (clock rate)

Each metric has a place and a purpose, and each can be misused

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr count	CPI	clock rate
Program			
Compiler			
Instr. Set			
Organization			
Technology			

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr count	CPI	clock rate
Program	X		
Compiler	X	X	
Instr. Set	X	X	X
Organization		X	X
Technology			X

CPI

"Average cycles per instruction"

CPI = (CPU Time * Clock Rate) / Instruction Count
= Clock Cycles / Instruction Count

$$\text{CPU time} = \text{ClockCycleTime} * \sum_{i=1}^{n} \text{CPI}_i * I_i$$

$$\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i * F_i \quad \text{where } F_i = \frac{I_i}{\text{Instruction Count}}$$

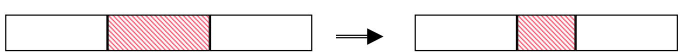
"instruction frequency"

Invest Resources where time is Spent!

Amdahl's Law

Speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}$$



Suppose that enhancement E accelerates a fraction F of the task by a factor S and the remainder of the task is unaffected then,

ExTime(with E) = ((1-F) + F/S) X ExTime(without E)

$$\text{Speedup(with E)} = \frac{1}{(1-F) + F/S}$$

Example (RISC processor)

Base Machine (Reg / Reg)

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%
			2.2	

Typical Mix

How much faster would the machine be is a better data cache reduced the average load time to 2 cycles?

How does this compare with using branch prediction to shave a cycle off the branch time?

What if two ALU instructions could be executed at once?

Evaluating Instruction Sets?

Design-time metrics:

° Can it be implemented, in how long, at what cost?

° Can it be programmed? Ease of compilation?

Static Metrics:

° How many bytes does the program occupy in memory?

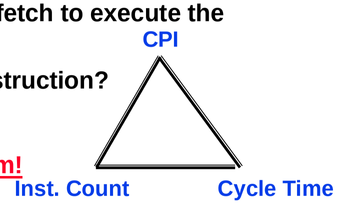
Dynamic Metrics:

° How many instructions are executed?

° How many bytes does the processor fetch to execute the program?

° How many clocks are required per instruction?

° How "lean" a clock is practical?



Best Metric: Time to execute the program!

NOTE: this depends on instructions set, processor organization, and compilation techniques.

Administrative Matters

° Unbalanced sections
  - Only 20 people in morning section!
  - Some way to rebalance?

° HW #2/Lab #2 out later tonight
  - "Broken" version of SPIM available later this week

° This lab will be done in pairs. Assigned in sections

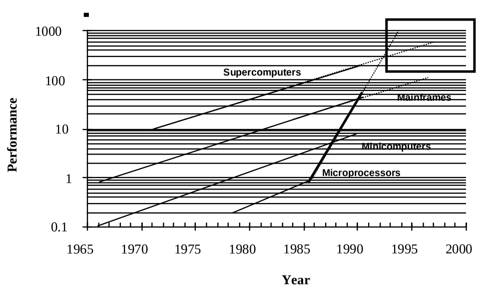
° Get Cory key card/card access to Cory 119

° Homework #1 due on Monday 2/1 at beginning of lecture

° Prerequisite quiz will also be on Monday: CS 61C, CS150 Review Chapters 1-4, (7.1-7.2), Ap, B of COD, Second Edition

° Lab 1 due Friday 1/29 by 5pm in box in 283 Soda Hall

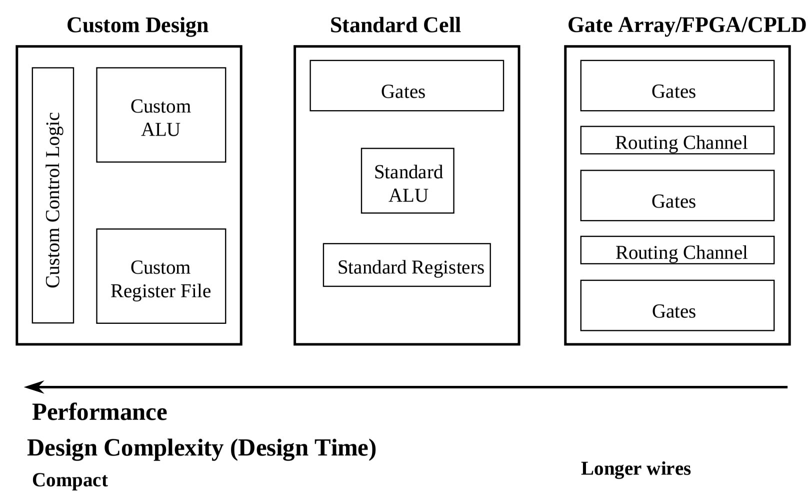
Performance and Technology Trends



° Technology Power: 1.2 x 1.2 x 1.2 = 1.7 x / year
  - Feature Size: shrinks 10% / yr. => Switching speed improves 1.2 / yr.
  - Density: improves 1.2x / yr.
  - Die Area: 1.2x / yr.

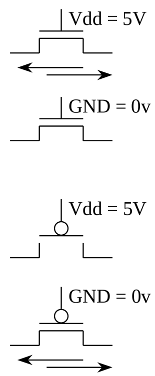
° The lesson of RISC is to keep the ISA as simple as possible:
  - Shorter design cycle => fully exploit the advancing technology (~3yr)
  - Advanced branch prediction and pipeline techniques
  - Bigger and more sophisticated on-chip caches

Range of Design Styles



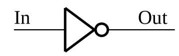
Basic Technology: CMOS

- CMOS: Complementary Metal Oxide Semiconductor
  - NMOS (N-Type Metal Oxide Semiconductor) transistors
  - PMOS (P-Type Metal Oxide Semiconductor) transistors
- NMOS Transistor
  - Apply a HIGH (Vdd) to its gate turns the transistor into a “conductor”
  - Apply a LOW (GND) to its gate shuts off the conduction path
- PMOS Transistor
  - Apply a HIGH (Vdd) to its gate shuts off the conduction path
  - Apply a LOW (GND) to its gate turns the transistor into a “conductor”

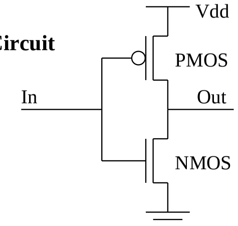


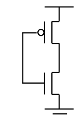
Basic Components: CMOS Inverter

Symbol

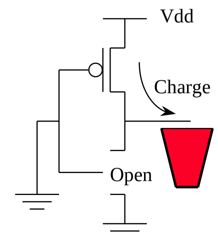


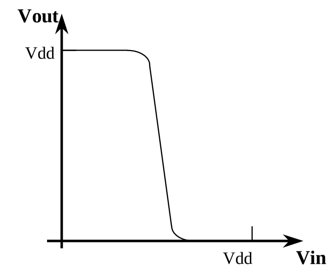
Circuit

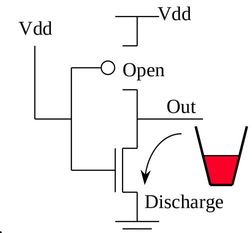




- Inverter Operation

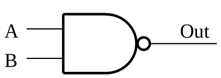




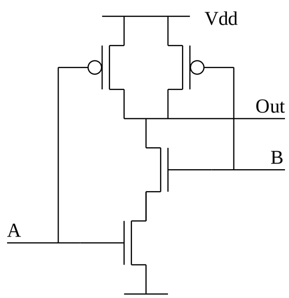


Basic Components: CMOS Logic Gates

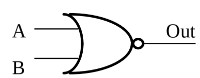
NAND Gate



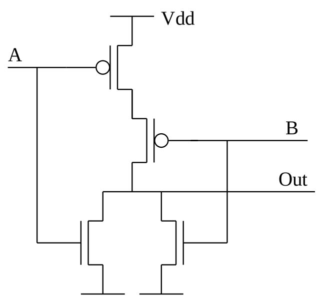
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0



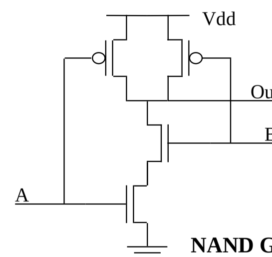
NOR Gate



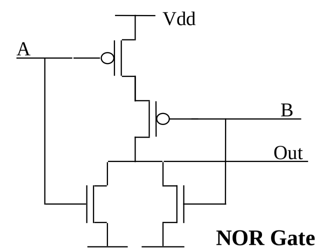
A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0



Gate Comparison



NAND Gate

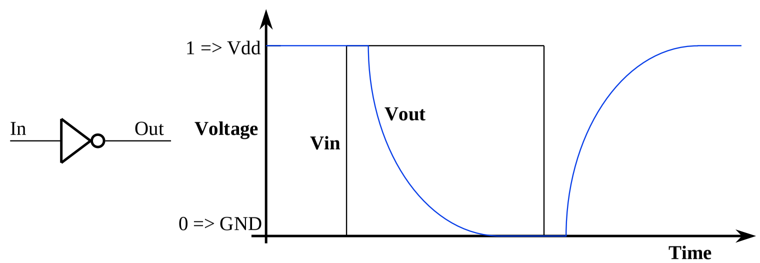


NOR Gate

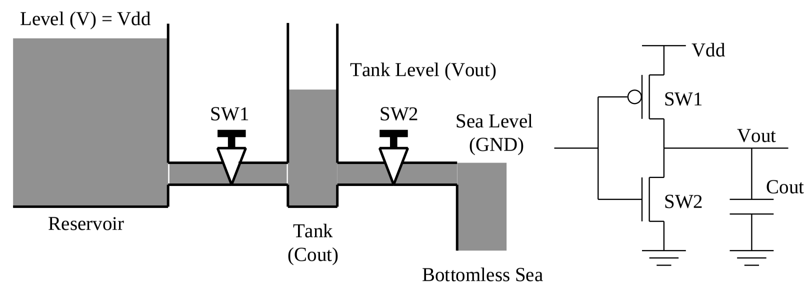
- If PMOS transistors is faster:
  - It is OK to have PMOS transistors in series
  - NOR gate is preferred
  - NOR gate is preferred also if H -> L is more critical than L -> H
- If NMOS transistors is faster:
  - It is OK to have NMOS transistors in series
  - NAND gate is preferred
  - NAND gate is preferred also if L -> H is more critical than H -> L

Ideal versus Reality

- When input 0 -> 1, output 1 -> 0 but NOT instantly
  - Output goes 1 -> 0: output voltage goes from Vdd (5v) to 0v
- When input 1 -> 0, output 0 -> 1 but NOT instantly
  - Output goes 0 -> 1: output voltage goes from 0v to Vdd (5v)
- Voltage does not like to change instantaneously

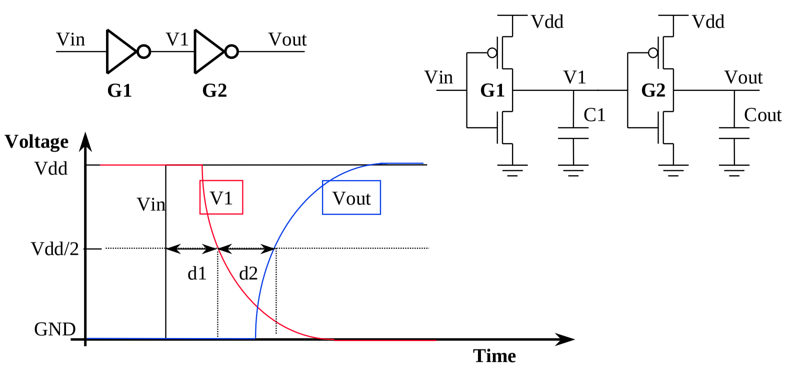


Fluid Timing Model



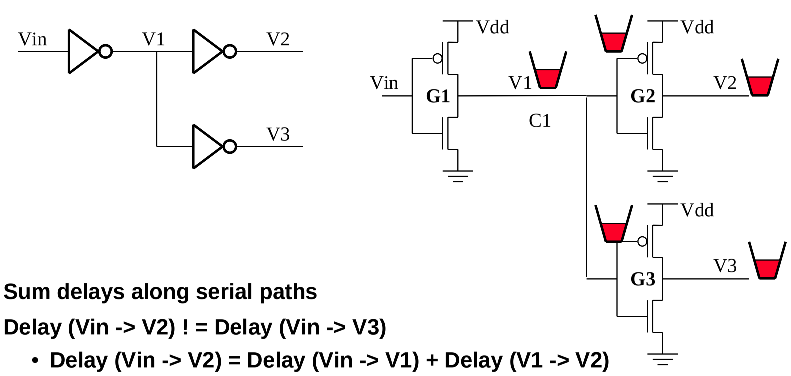
- Water <-> Electrical Charge   Tank Capacity <-> Capacitance (C)
- Water Level <-> Voltage   Water Flow <-> Charge Flowing (Current)
- Size of Pipes <-> Strength of Transistors (G)
- Time to fill up the tank proportional to C / G

Series Connection



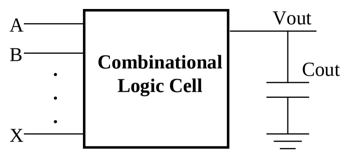
- Total Propagation Delay = Sum of individual delays = d1 + d2
- Capacitance C1 has two components:
  - Capacitance of the wire connecting the two gates
  - Input capacitance of the second inverter

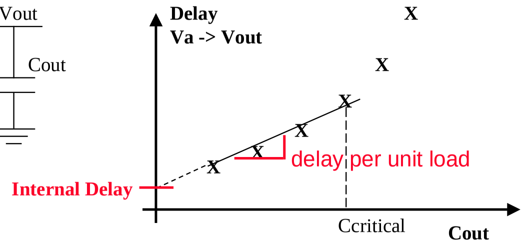
Review: Calculating Delays



- Sum delays along serial paths
- Delay (Vin -> V2) ! = Delay (Vin -> V3)
  - Delay (Vin -> V2) = Delay (Vin -> V1) + Delay (V1 -> V2)
  - Delay (Vin -> V3) = Delay (Vin -> V1) + Delay (V1 -> V3)
- Critical Path = The longest among the N parallel paths
- C1 = Wire C + Cin of Gate 2 + Cin of Gate 3

Review: General C/L Cell Delay Model

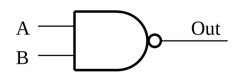




- Combinational Cell (symbol) is fully specified by:
  - functional (input -> output) behavior
    - truth-table, logic equation, VHDL
  - load factor of each input
  - critical propagation delay from each input to each output for each transition
    - T_{HL}(A, o) = Fixed Internal Delay + Load-dependent-delay x *load*
- Linear model composes

Characterize a Gate

- Input capacitance for each input
- For each input-to-output path:
  - For each output transition type (H->L, L->H, H->Z, L->Z ... etc.)
    - Internal delay (ns)
    - Load dependent delay (ns / fF)
- Example: 2-input NAND Gate

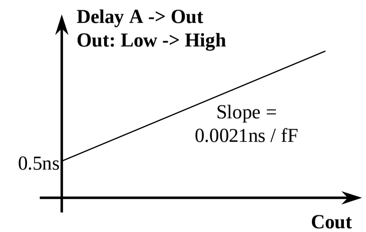


For A and B: Input Load (I.L.) = 61 fF

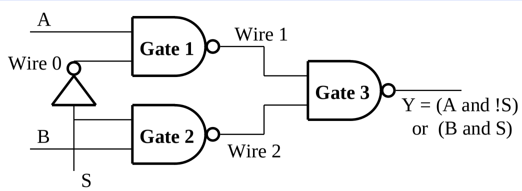
For either A -> Out or B -> Out:

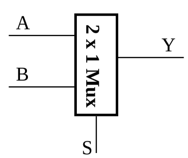
Tlh = 0.5ns Tlhf = 0.0021ns / fF

Thl = 0.1ns Thlf = 0.0020ns / fF



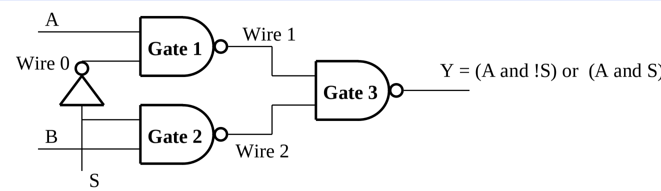
A Specific Example: 2 to 1 MUX





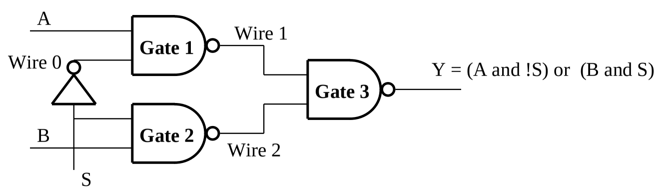
- Input Load (I.L.)
  - A, B: I.L. (NAND) = 61 fF
  - S: I.L. (INV) + I.L. (NAND) = 50 fF + 61 fF = 111 fF
- Load Dependent Delay (L.D.D.): Same as Gate 3
  - TAYlhf = 0.0021 ns / fF   TAYhlf = 0.0020 ns / fF
  - TBYlhf = 0.0021 ns / fF   TBYhlf = 0.0020 ns / fF
  - TSYlhf = 0.0021 ns / fF   TSYlhf = 0.0020 ns / fF

2 to 1 MUX: Internal Delay Calculation



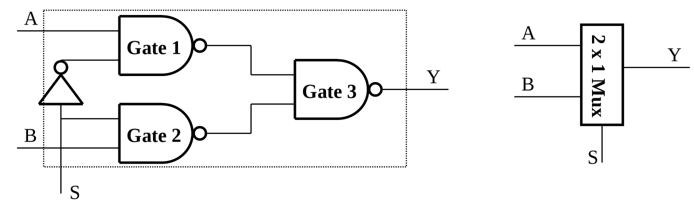
- Internal Delay (I.D.):
  - A to Y: I.D. G1 + (Wire 1 C + G3 Input C) * L.D.D G1 + I.D. G3
  - B to Y: I.D. G2 + (Wire 2 C + G3 Input C) * L.D.D. G2 + I.D. G3
  - S to Y (Worst Case) : I.D. Inv + (Wire 0 C + G1 Input C) * L.D.D. Inv + Internal Delay A to Y
- We can approximate the effect of "Wire 1 C" by:
  - Assume Wire 1 has the same C as all the gate C attached to it.

2 to 1 MUX: Internal Delay Calculation (continue)



- Internal Delay (I.D.):
  - A to Y: I.D. G1 + (Wire 1 C + G3 Input C) * L.D.D G1 + I.D. G3
  - B to Y: I.D. G2 + (Wire 2 C + G3 Input C) * L.D.D. G2 + I.D. G3
  - S to Y (Worst Case): I.D. Inv + (Wire 0 C + G1 Input C) * L.D.D. Inv + Internal Delay A to Y
- Specific Example:
  - TAYlh = TPhl G1 + (2.0 * 61 fF) * TPhlf G1 + TPlh G3
    = 0.1ns + 122 fF * 0.0020 ns/fF + 0.5ns = 0.844 ns

Abstraction: 2 to 1 MUX



- Input Load: A = 61 fF, B = 61 fF, S = 111 fF
- Load Dependent Delay:
  - TAYlhf = 0.0021 ns / fF   TAYhlf = 0.0020 ns / fF
  - TBYlhf = 0.0021 ns / fF   TBYhlf = 0.0020 ns / fF
  - TSYlhf = 0.0021 ns / fF   TSYlhf = 0.0020 ns / f F
- Internal Delay:
  - TAYlh = TPhl G1 + (2.0 * 61 fF) * TPhlf G1 + TPlh G3
    = 0.1ns + 122 fF * 0.0020ns/fF + 0.5ns = 0.844ns
  - Fun Exercises: TAYhl, TBYlh, TSYlh, TSYlh

Break (5 Minutes)

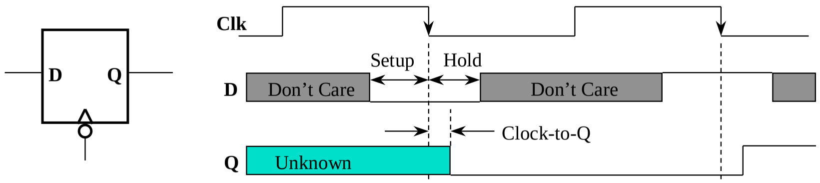
CS152 Logic Elements

- NAND2, NAND3, NAND 4
- NOR2, NOR3, NOR4
- INV1x (normal inverter)
- INV4x (inverter with large output drive)

- XOR2
- XNOR2
- PWR: Source of 1's
- GND: Source of 0's
- fast MUXes (maybe)

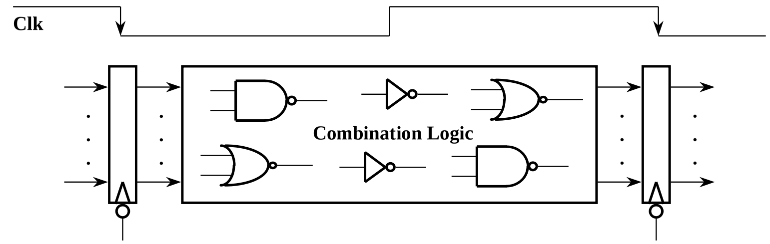
- D flip flop with negative edge triggered

Storage Element's Timing Model



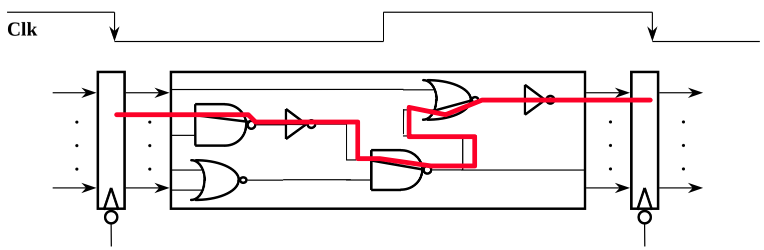
- Setup Time: Input must be stable BEFORE the trigger clock edge
- Hold Time: Input must REMAIN stable after the trigger clock edge
- Clock-to-Q time:
  - Output cannot change instantaneously at the trigger clock edge
  - Similar to delay in logic gates, two components:
    - Internal Clock-to-Q
    - Load dependent Clock-to-Q
- Typical for class: 1ns Setup, 0.5ns Hold

Clocking Methodology



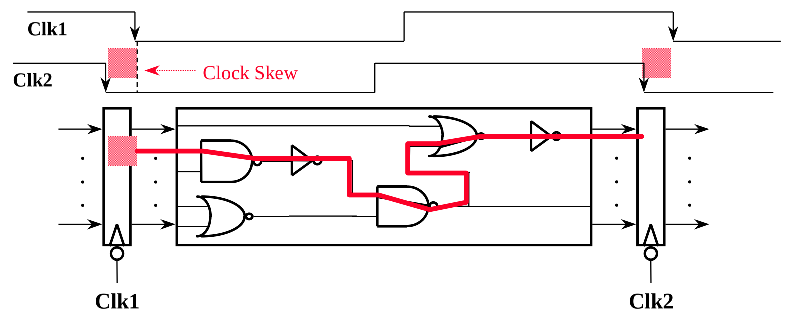
- All storage elements are clocked by the same clock edge
- The combination logic block's:
  - Inputs are updated at each clock tick
  - All outputs MUST be stable before the next clock tick

Critical Path & Cycle Time



- Critical path: the slowest path between any two storage devices
- Cycle time is a function of the critical path
- must be greater than:
  - Clock-to-Q + Longest Path through Combination Logic + Setup

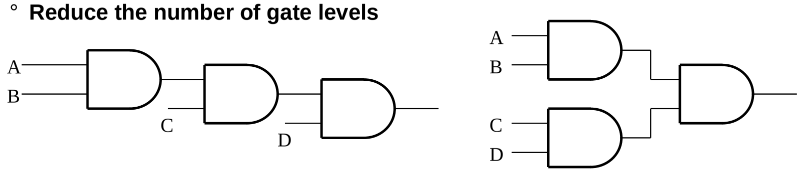
Clock Skew's Effect on Cycle Time



- The worst case scenario for cycle time consideration:
  - The input register sees CLK1
  - The output register sees CLK2
- Cycle Time - Clock Skew ≥ CLK-to-Q + Longest Delay + Setup
  ⇒ Cycle Time ≥ CLK-to-Q + Longest Delay + Setup + Clock Skew

Tricks to Reduce Cycle Time

° Reduce the number of gate levels

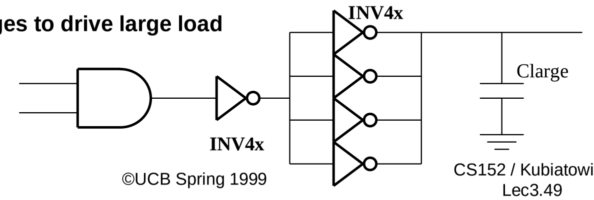


° Pay attention to loading

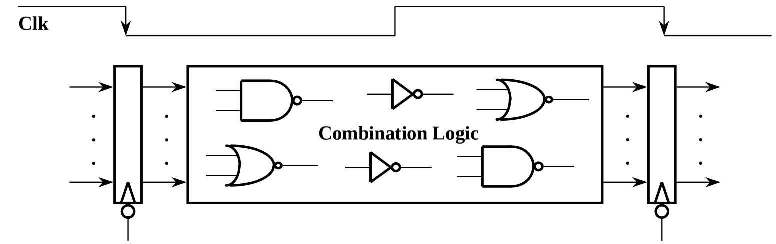
° One gate driving many gates is a bad idea

° Avoid using a small gate to drive a long wire

° Use multiple stages to drive large load



How to Avoid Hold Time Violation?



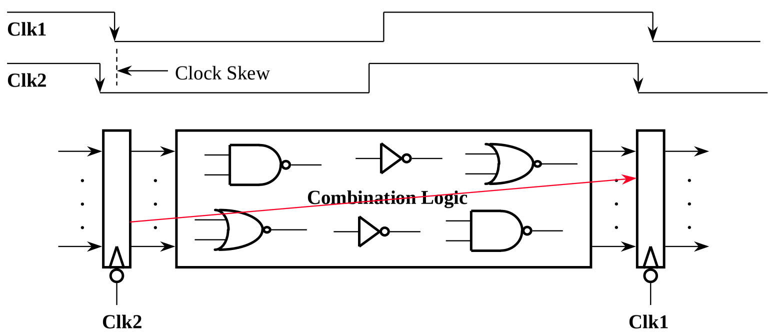
° Hold time requirement:
  • Input to register must NOT change immediately after the clock tick

° This is usually easy to meet in the "edge trigger" clocking scheme

° Hold time of most FFs is <= 0 ns

° CLK-to-Q + Shortest Delay Path must be greater than Hold Time

Clock Skew's Effect on Hold Time



° The worst case scenario for hold time consideration:
  • The input register sees CLK2
  • The output register sees CLK1
  • fast FF2 output must not change input to FF1 for same clock edge

° (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time

Summary

° Total execution time is the most reliable measure of performance

° Amdall's law: Law of Diminishing Returns

° Performance and Technology Trends
  • Keep the design simple (KISS rule) to take advantage of the latest technology
  • CMOS inverter and CMOS logic gates

° Delay Modeling and Gate Characterization
  • Delay = Internal Delay + (Load Dependent Delay x Output Load)

° Clocking Methodology and Timing Considerations
  • Simplest clocking methodology
    - All storage elements use the SAME clock edge
  • Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
  • (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time

To Get More Information

- **A Classic Book that Started it All:**
  - **Carver Mead and Lynn Conway, “Introduction to VLSI Systems,” Addison-Wesley Publishing Company, October 1980.**
- **A Good VLSI Circuit Design Book**
  - **Lance Glasser & Daniel Dobberpuhl, “The Design and Analysis of VLSI Circuits,” Addison-Wesley Publishing Company, 1985.**
    - **Mr. Dobberpuhl is responsible for the DEC Alpha chip design.**
- **A Book on How and Why Digital ICs Work:**
  - **David Hodges & Horace Jackson, “Analysis and Design of Digital Integrated Circuits,” McGraw-Hill Book Company, 1983.**
- **New Book:**
  - **Jan Rabaey, “Digital Integrated Circuits: A Design Perspective,” Prentice-Hall Publishers, 1998.**