inst.eecs.berkeley.edu/~cs61c/su06

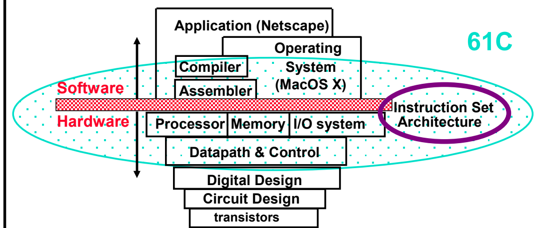
CS61C : Machine Structures

Lecture #14: Combinational Logic, Gates, and State

2006-07-20

Andy Carle

What are “Machine Structures”?



Coordination of many *levels of abstraction*

We’ll investigate lower abstraction layers! (contract between HW & SW)

Below the Program

- High-level language program (in C)

```
swap  int v[], int k){
      int temp;
      temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
}
```

C compiler

- Assembly language program (for MIPS)

```
swap: sll   $2, $5, 2
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

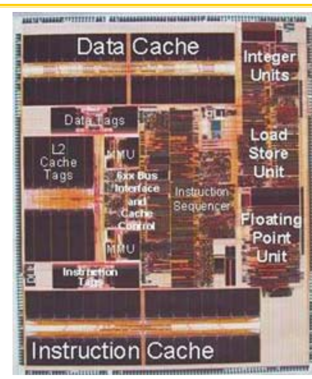
assembler

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000 . . .
```

?

Physical Hardware - PowerPC 750



Digital Design Basics (1/2)

- Next 2 weeks: we’ll study how a modern processor is built starting with basic logic elements as building blocks.
- Why study logic design?
  - Understand what processors can do fast and what they can’t do fast (avoid slow things if you want your code to run fast!)
  - Background for more detailed hardware courses (CS 150, CS 152)

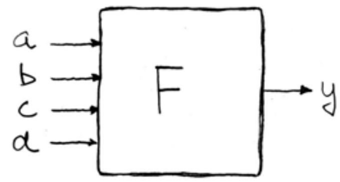
Digital Design Basics (2/2)

- ISA is very important abstraction layer
  - Contract between HW and SW
  - Can you peek across abstraction?
  - Can you depend “across abstraction”?
- Voltages are analog, quantized to 0/1
- Circuit delays are fact of life
- Two types
  - Stateless Combinational Logic (&,|,~)
  - State circuits (e.g., registers)

Outline

- Truth Tables
- Transistors
- Logic Gates
- Combinational Logic
- Boolean Algebra

Truth Tables (1/6)

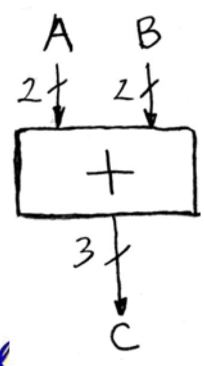


a	b	c	d	y
0	0	0	0	F(0,0,0,0)
0	0	0	1	F(0,0,0,1)
0	0	1	0	F(0,0,1,0)
0	0	1	1	F(0,0,1,1)
0	1	0	0	F(0,1,0,0)
0	1	0	1	F(0,1,0,1)
0	1	1	0	F(0,1,1,0)
0	1	1	1	F(0,1,1,1)
1	0	0	0	F(1,0,0,0)
1	0	0	1	F(1,0,0,1)
1	0	1	0	F(1,0,1,0)
1	0	1	1	F(1,0,1,1)
1	1	0	0	F(1,1,0,0)
1	1	0	1	F(1,1,0,1)
1	1	1	0	F(1,1,1,0)
1	1	1	1	F(1,1,1,1)

TT (2/6) Ex #1: 1 iff one (not both) a,b=1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

TT (3/6): Example #2: 2-bit adder



A a_1a_0	B b_1b_0	C $c_2c_1c_0$
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

TT (4/6): Ex #3: 32-bit unsigned adder

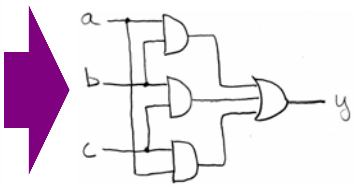
A	B	C
000 ... 0	000 ... 0	000 ... 00
000 ... 0	000 ... 1	000 ... 01
.	.	.
.	.	.
.	.	.
111 ... 1	111 ... 1	111 ... 10

TT (5/6): Conversion: 3-input majority

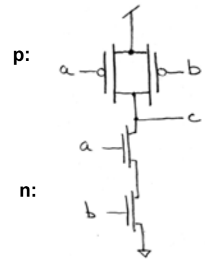
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

TT (6/6): Conversion: 3-input majority

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Transistors (1/3)



CMOSFET Transistors:

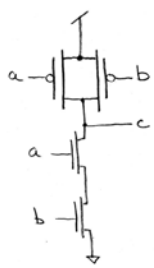
* Physically exist!

* Voltages are quantized

* Only 2 Types:
- P-channel:
0 on gate -> pull up (1)
- N-channel:
1 on gate -> pull down (0)

* Undriven otherwise.

Transistors (2/3)



CMOSFET Transistors:

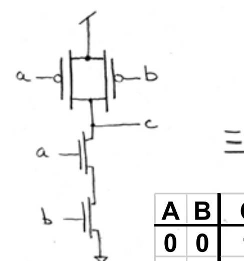
* have delay and require power

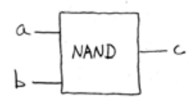
* can be combined to perform logical operations and maintain state.

- logical operations will be our starting point for digital design

- state tomorrow

Transistors (3/3): CMOS ➔ Nand





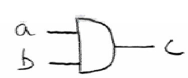
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Logic Gates (1/4)

- Transistors are too low level
  - Good for measuring performance, power.
  - Bad for logical design / analysis
- Gates are collections of transistors wired in a certain way
  - Can represent and reason about gates with truth tables and Boolean algebra
  - We will mainly review the concepts of truth tables and Boolean algebra in this class. It is assumed that you've seen these before.
  - Section B.2 in the textbook has a review

Logic Gates (2/4)

AND

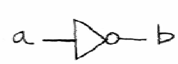


ab	c
00	0
01	0
10	0
11	1

OR

ab	c
00	0
01	1
10	1
11	1

NOT

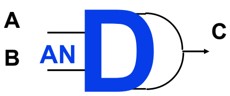


a	b
0	1
1	0

Logic Gates (3/4)

AND Gate

Symbol



Definition

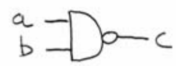
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Logic Gates (4/4)

XOR

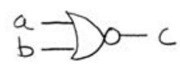
ab	c
00	0
01	1
10	1
11	0

NAND



ab	c
00	1
01	1
10	1
11	0

NOR



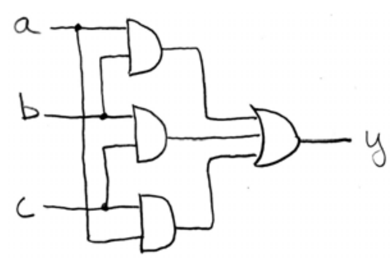
ab	c
00	1
01	0
10	0
11	0

Boolean Algebra (1/7)

- George Boole, 19th Century mathematician
- Developed a mathematical system (algebra) involving logic, later known as "Boolean Algebra"
- Primitive functions: AND, OR and NOT
- The power of BA is there's a one-to-one correspondence between circuits made up of AND, OR and NOT gates and equations in BA

\+ means OR,• means AND, $\overline{x}$ means NOT

BA (2/7): e.g., majority circuit



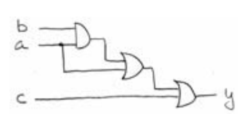
y = a • b + a • c + b • c

y = ab + ac + bc

BA (3/7):Laws of Boolean Algebra

$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$	complementarity
$x \cdot 0 = 0$	$x + 1 = 1$	laws of 0's and 1's
$x \cdot 1 = x$	$x + 0 = x$	identities
$x \cdot x = x$	$x + x = x$	idempotent law
$x \cdot y = y \cdot x$	$x + y = y + x$	commutativity
$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$	associativity
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$	distribution
$xy + x = x$	$(x + y)x = x$	uniting theorem
$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{(x + y)} = \overline{x} \cdot \overline{y}$	DeMorgan's Law

BA (4/7): Circuit & Algebraic Simplification



original circuit

↓

$y = ((ab) + a) + c$ — equation derived from original circuit

↓

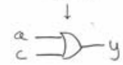
$= ab + a + c$ — algebraic simplification

$= a(b + 1) + c$

$= a(1) + c$

$= a + c$

↓



simplified circuit

BA (5/7): Simplification Example

$$
\begin{aligned}
y &= ab + a + c \\
&= a(b+1) + c && \textit{distribution, identity} \\
&= a(1) + c && \textit{law of 1's} \\
&= a + c && \textit{identity}
\end{aligned}
$$

BA (6/7): Canonical forms (1/2)

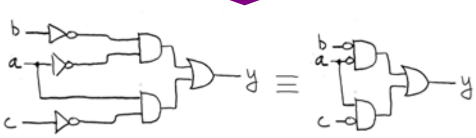
	abc	y
$\bar{a}\cdot\bar{b}\cdot\bar{c}$	000	1
$\bar{a}\cdot\bar{b}\cdot c$	001	1
	010	0
	011	0
$a\cdot\bar{b}\cdot\bar{c}$	100	1
	101	0
$a\cdot b\cdot\bar{c}$	110	1
	111	0

Sum-of-products (ORs of ANDs)

$$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}$$

BA (7/7): Canonical forms (2/2)

$$
\begin{aligned}
y &= \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c} \\
&= \bar{a}\bar{b}(\bar{c} + c) + a\bar{c}(\bar{b} + b) && \textit{distribution} \\
&= \bar{a}\bar{b}(1) + a\bar{c}(1) && \textit{complementarity} \\
&= \bar{a}\bar{b} + a\bar{c} && \textit{identity}
\end{aligned}
$$



Combinational Logic

A *combinational* logic block is one in which the output is a function only of its current input.

- **Combinational logic cannot have memory.**
- **Everything we've seen so far is CL**
- **CL will have delay (f(transistors))**

Peer Instruction

A. (a+b)• (a+b) = b

B. N-input gates can be thought of as cascaded 2-input gates. I.e., (a ∆ bc ∆ d ∆ e) = a ∆ (bc ∆ (d ∆ e)) where ∆ is one of AND, OR, XOR, NAND

C. You can use NOR(s) with clever wiring to simulate AND, OR, & NOT

Administrivia

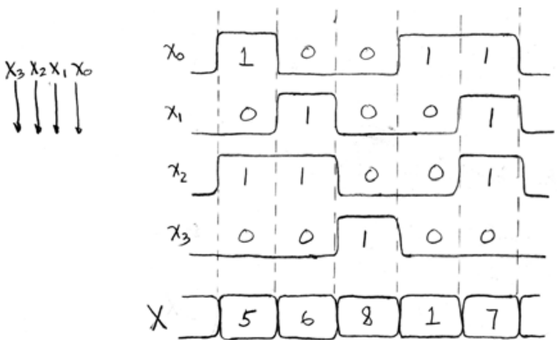
- **HW 4 due Friday**
- **Project 2 due Friday the 28th**
- **If you want to get a little bit ahead (in a moderately fun sort of way), start playing with Logisim:**
  - **http://ozark.hendrix.edu/~burch/logisim/**

Signals and Waveforms

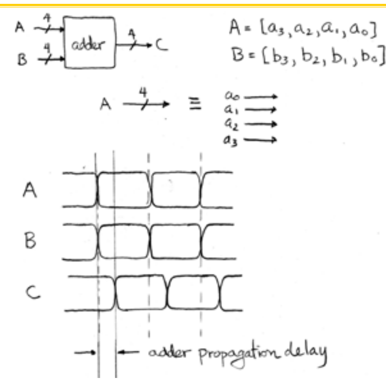
- Outputs of CL change over time
  - With what? → Change in inputs
  - Can graph changes with waveforms …

Signals and Waveforms: Adders

Signals and Waveforms: Grouping



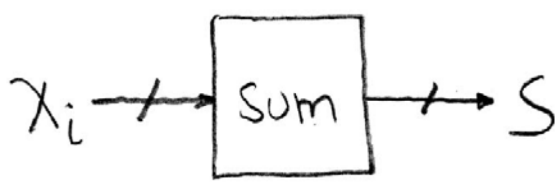
Signals and Waveforms: Circuit Delay



State

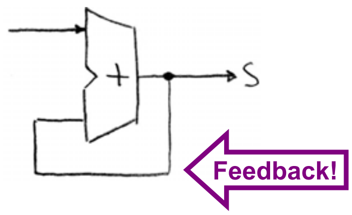
- With CL, output is always a function of CURRENT input
  - With some (variable) propagation delay
- Clearly, we need a way to introduce state into computation

Accumulator Example



Want: `S=0; for i from 0 to n-1`
`S = S + X`$_i$

First try…Does this work?



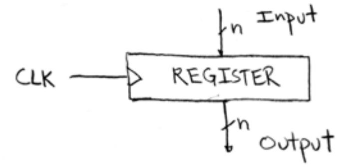
Nope!

Reason #1… What is there to control the next iteration of the 'for' loop?

Reason #2… How do we say: 'S=0'?

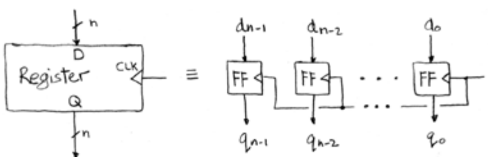
Need a way to store partial sums! …

Circuits with STATE (e.g., register)



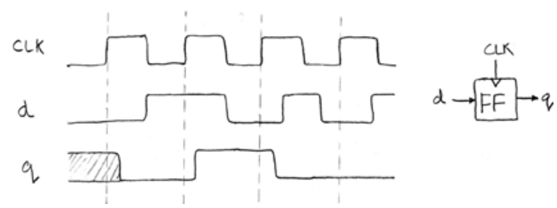
Need a Logic Block that will:
1. store output (partial sum) for a while,
2. until we tell it to update with a new value.

Register Details…What's in it anyway?



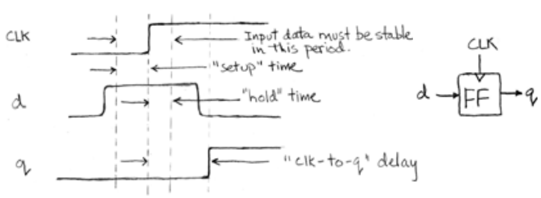
- **n instances of a "Flip-Flop", called that because the output flips and flops betw. 0,1**
- **D is "data"**
- **Q is "output"**
- **Also called "d-q Flip-Flop","d-type Flip-Flop"**

What's the timing of a Flip-flop? (1/2)



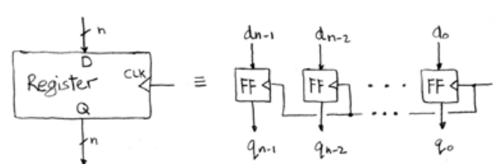
- **Edge-triggered D-type flip-flop**
  - **This one is "positive edge-triggered"**
- **"On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."**

What's the timing of a Flip-flop? (2/2)



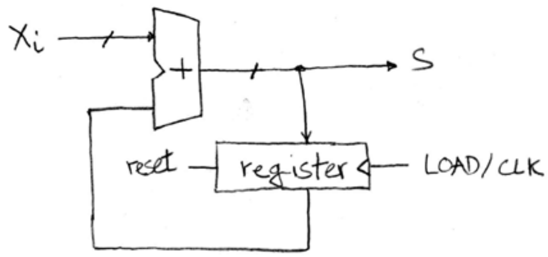
- **Edge-triggered D-type flip-flop**
  - **This one is "positive edge-triggered"**
- **"On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."**

Bus a bunch of D FFs together …

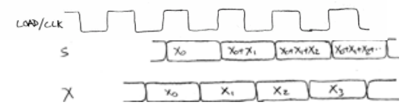


- **Register of size N:**
  - **n instances of D Flip-Flop**

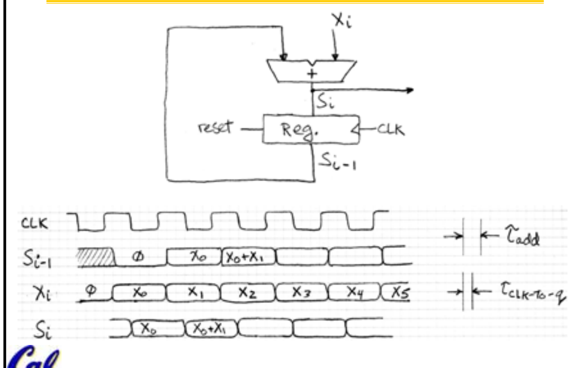
Second try…How about this? Yep!



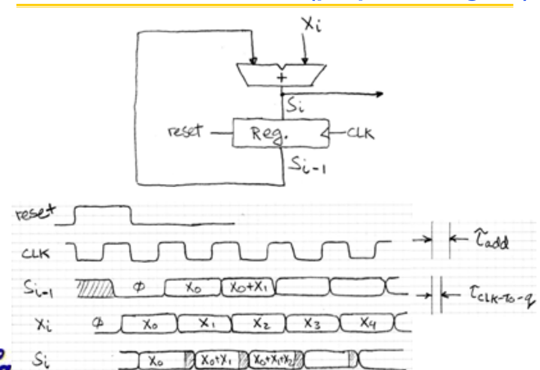
Rough timing…



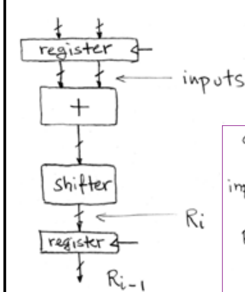
Accumulator Revisited (proper timing 1/2)



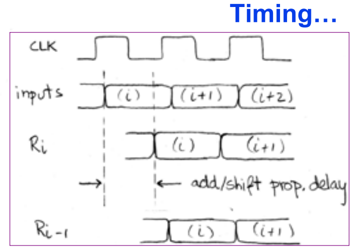
Accumulator Revisited (proper timing 2/2)



Pipelining to improve performance (1/2)

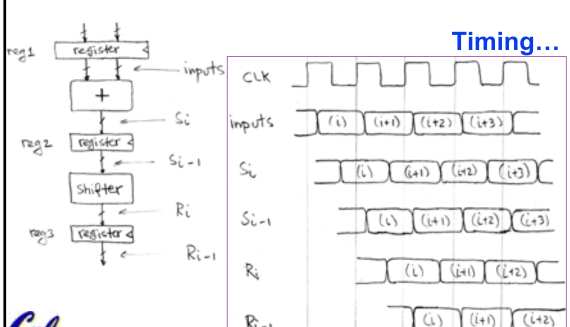


Timing…



Pipelining to improve performance (2/2)

Timing…



Peer Instruction 2

- Simplify the following Boolean algebra equation:
- Q = !(A*B) + !(!A * C)
- Use algebra, individual steps, etc.
  - Don't just look at it and figure it out, or I'll have to start using harder examples. ☺

"And In conclusion…"

- Use this table and techniques we learned to transform from 1 to another