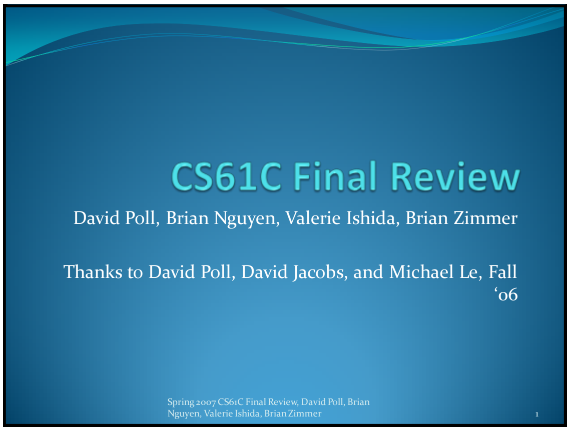
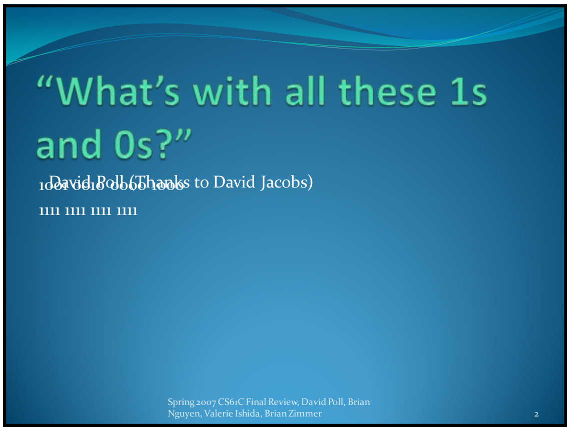


CS61C Final Review

David Poll, Brian Nguyen, Valerie Ishida, Brian Zimmer

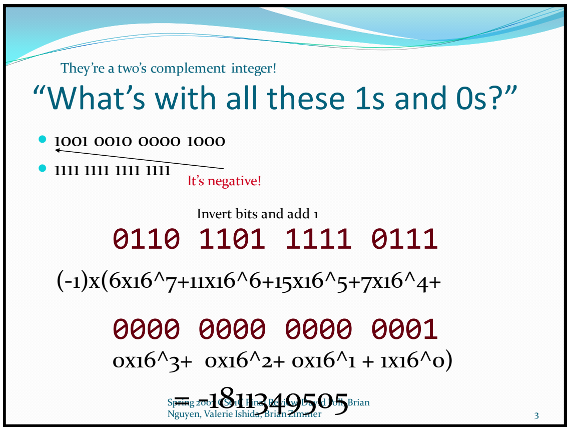
Thanks to David Poll, David Jacobs, and Michael Le, Fall '06



"What's with all these 1s and 0s?"

1001 0010 0000 1000
David Poll (Thanks to David Jacobs)

1111 1111 1111 1111



They're a two's complement integer!

"What's with all these 1s and 0s?"

- 1001 0010 0000 1000
- 1111 1111 1111 1111

It's negative!

Invert bits and add 1

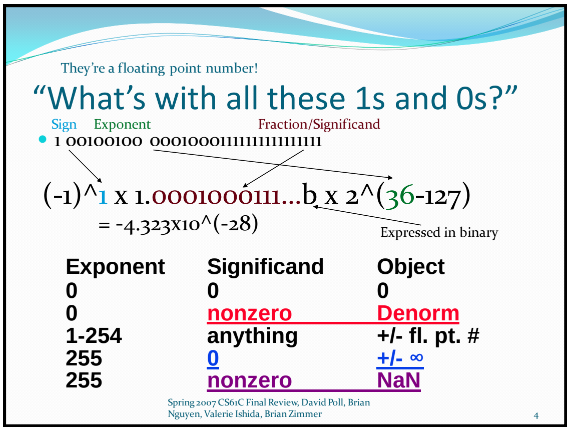
0110 1101 1111 0111

(-1)x(6x16^7+11x16^6+15x16^5+7x16^4+

0000 0000 0000 0001

0x16^3+ 0x16^2+ 0x16^1 + 1x16^0)

= -1811349505



They're a floating point number!

"What's with all these 1s and 0s?"

Sign Exponent Fraction/Significand

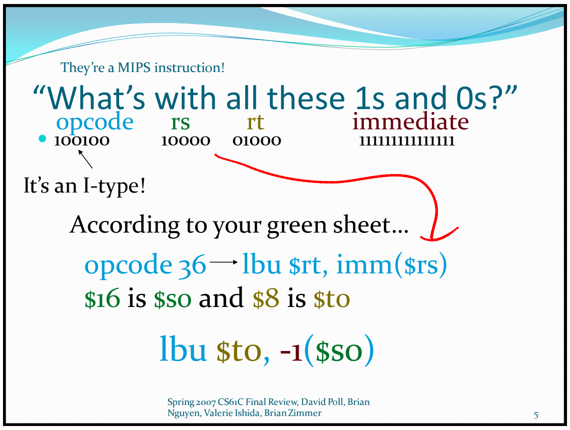
- 1 00100100 0001000111111111111111

(-1)^1 x 1.0001000111…b x 2^(36-127)

= -4.323x10^(-28)

Expressed in binary

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN



They're a MIPS instruction!

"What's with all these 1s and 0s?"

opcode rs rt immediate

- 100100 10000 01000 1111111111111111

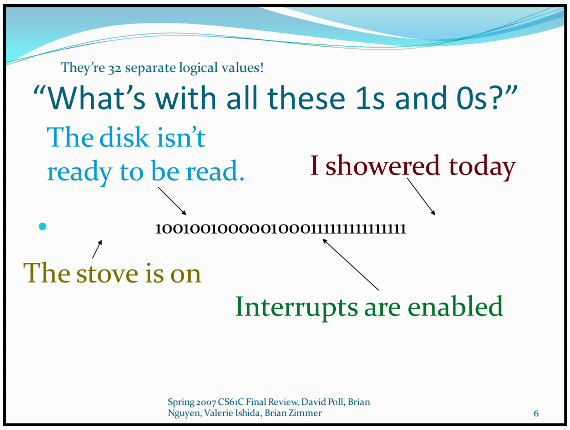
It's an I-type!

According to your green sheet…

opcode 36 → lbu $rt, imm($rs)

$16 is $s0 and $8 is $t0

lbu $t0, -1($s0)



They're 32 separate logical values!

"What's with all these 1s and 0s?"

The disk isn't ready to be read.

I showered today

- 10010010000010001111111111111111

The stove is on

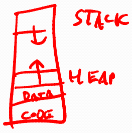
Interrupts are enabled

If there's one thing you learn…

N bits can represent 2^N things

C and Memory

- Get an n-element array of things
- array = (thing *)
- malloc(n*sizeof(thing));



- Don't forget to free it later.
- free(array);

Problem!

- typedef struct node {
- int value;
- struct node* next;
- } ent;

// reserve space

- stack push(stack s,int val){

// set values

// return new node

- }

```
typedef ent * stack;

int peek(stack s){
    //return value
}

stack pop(stack s,int * val){
    // change →
    // free entry on top
    // return the next one

}
```

Problem!

- typedef struct node {
- int value;
- struct node* next;
- } ent;

- stack push(stack s,int val){
- ent * new = (ent *)
- malloc (sizeof(ent));
- new->value = val;
- new->next = s;
- return new;
- }

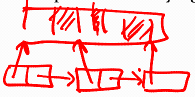
```
typedef ent * stack;

int peek(stack s){
   return s->value;
}

stack pop(stack s, int * val){
   ent * temp = s->next;
   *val = s->value;
   free(s);
   return temp;
}
```

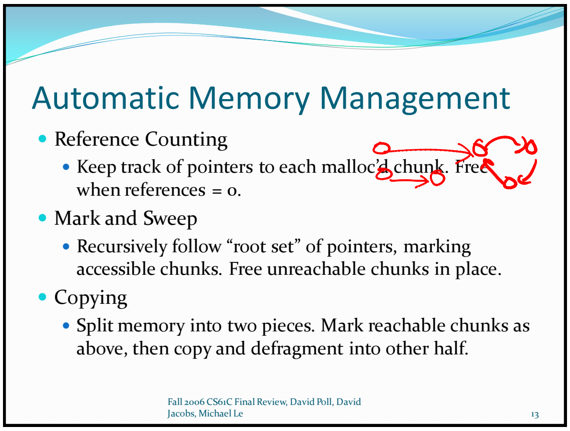
Memory Management

- First fit
  - Allocate the first available chunk big enough
- Next fit
  - Allocate the first chunk after the last one allocated
- Best fit
  - Allocate the smallest chunk capable of satisfying the request



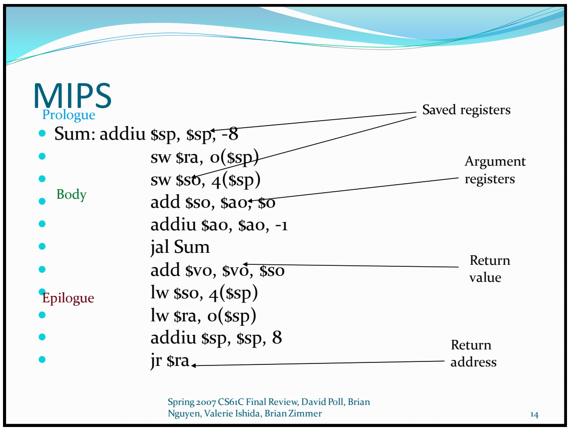
Memory Management

- Free List
  - Linked list of free chunks, use first/next/best fit
- Slab Allocator
  - Fixed number of 2^n sized chunks, can use a bitmap to track. Free list for larger requests.
- Buddy Allocator
  - 2^n chunks can merge with their "buddy" to make a 2^(n+1) chunk. Free list for larger requests.



Automatic Memory Management

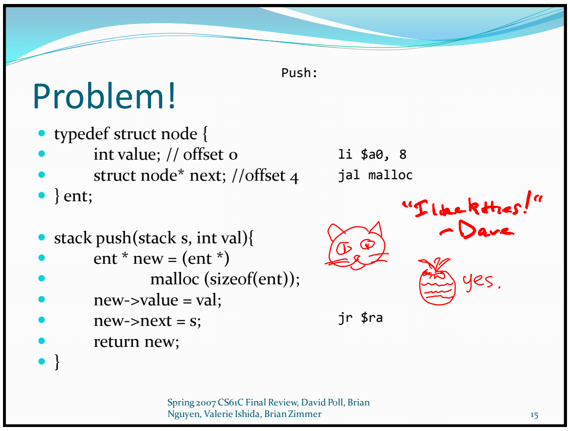
- Reference Counting
  - Keep track of pointers to each malloc'd chunk. Free when references = 0.
- Mark and Sweep
  - Recursively follow "root set" of pointers, marking accessible chunks. Free unreachable chunks in place.
- Copying
  - Split memory into two pieces. Mark reachable chunks as above, then copy and defragment into other half.



MIPS

Prologue

```
Sum: addiu $sp, $sp, -8
             sw $ra, 0($sp)
             sw $s0, 4($sp)
             add $s0, $a0, $0
             addiu $a0, $a0, -1
             jal Sum
             add $v0, $v0, $s0
             lw $s0, 4($sp)
             lw $ra, 0($sp)
             addiu $sp, $sp, 8
             jr $ra
```



Problem!

```
typedef struct node {
        int value; // offset 0
        struct node* next; //offset 4
} ent;

stack push(stack s, int val){
        ent * new = (ent *)
                malloc (sizeof(ent));
        new->value = val;
        new->next = s;
        return new;
}
```

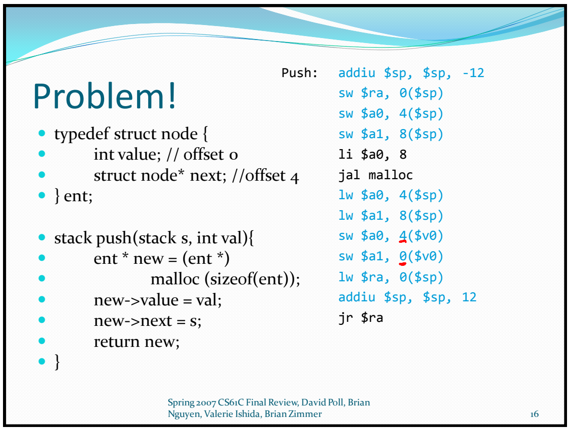
```
Push:

li $a0, 8
jal malloc

jr $ra
```

"I like kitties!" - Dave

yes.

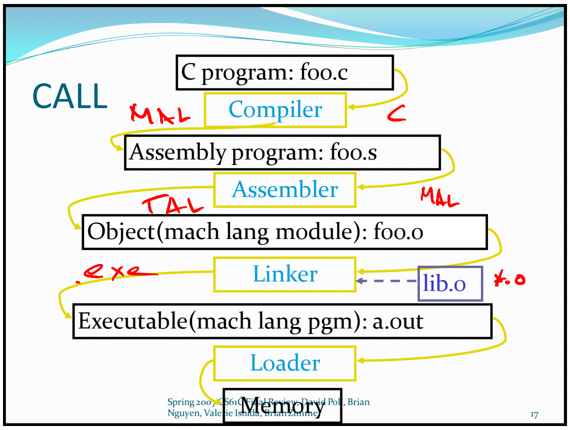


Problem!

```
typedef struct node {
        int value; // offset 0
        struct node* next; //offset 4
} ent;

stack push(stack s, int val){
        ent * new = (ent *)
                malloc (sizeof(ent));
        new->value = val;
        new->next = s;
        return new;
}
```

```
Push:   addiu $sp, $sp, -12
        sw $ra, 0($sp)
        sw $a0, 4($sp)
        sw $a1, 8($sp)
        li $a0, 8
        jal malloc
        lw $a0, 4($sp)
        lw $a1, 8($sp)
        sw $a0, 4($v0)
        sw $a1, 0($v0)
        lw $ra, 0($sp)
        addiu $sp, $sp, 12
        jr $ra
```



CALL

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object(mach lang module): foo.o

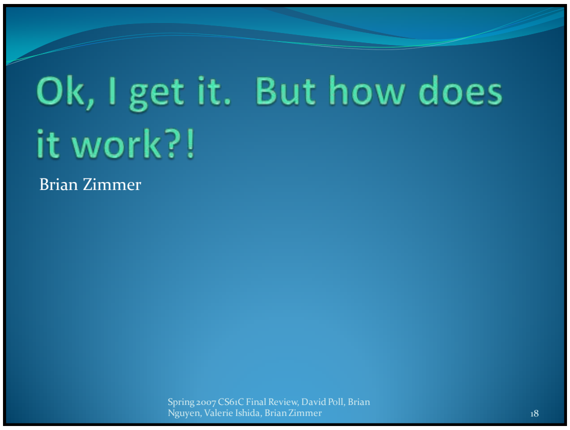
Linker

lib.o

Executable(mach lang pgm): a.out

Loader

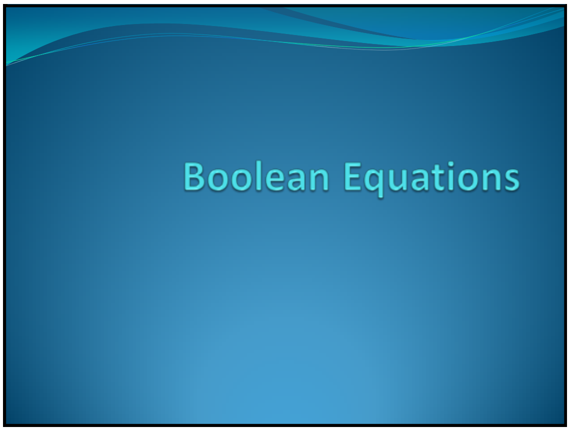
Memory



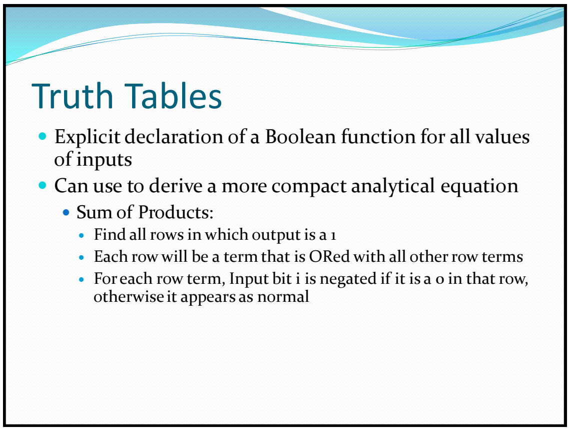
Ok, I get it. But how does it work?!

Brian Zimmer

SDS Review

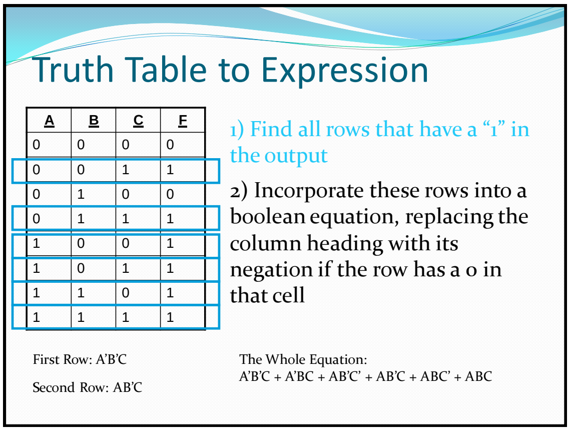


Boolean Equations



Truth Tables

- Explicit declaration of a Boolean function for all values of inputs
- Can use to derive a more compact analytical equation
  - Sum of Products:
    - Find all rows in which output is a 1
    - Each row will be a term that is ORed with all other row terms
    - For each row term, Input bit i is negated if it is a 0 in that row, otherwise it appears as normal



Truth Table to Expression

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

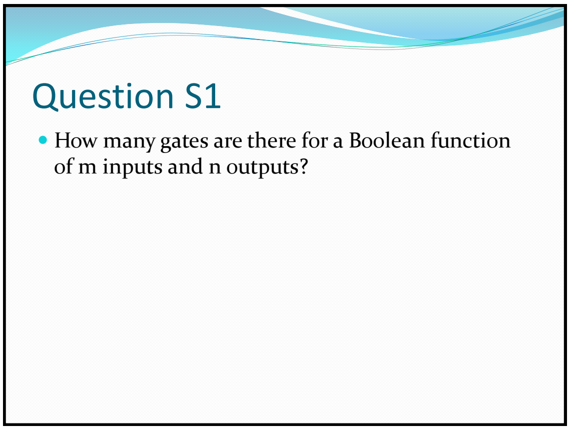
1) Find all rows that have a “1” in the output

2) Incorporate these rows into a boolean equation, replacing the column heading with its negation if the row has a 0 in that cell

First Row: A'B'C

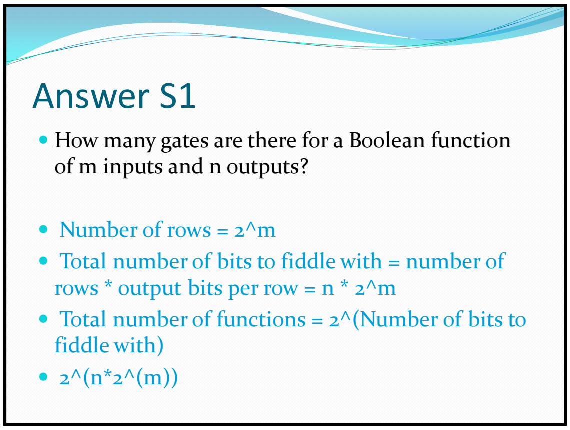
Second Row: AB'C

The Whole Equation:
A'B'C + A'BC + AB'C' + AB'C + ABC' + ABC



Question S1

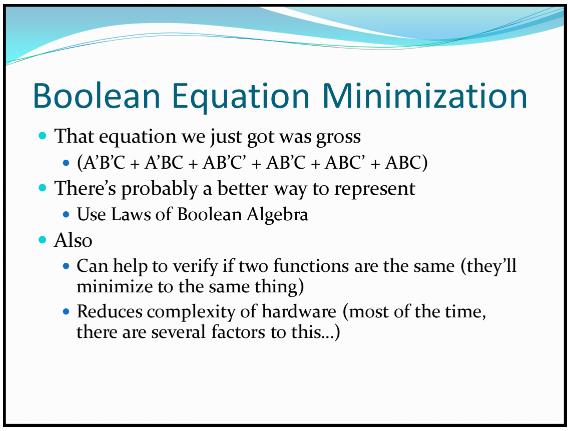
- How many gates are there for a Boolean function of m inputs and n outputs?



Answer S1

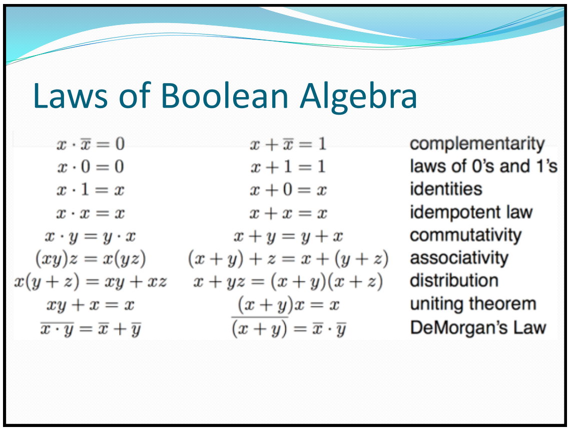
- How many gates are there for a Boolean function of m inputs and n outputs?

- Number of rows = 2^m
- Total number of bits to fiddle with = number of rows * output bits per row = n * 2^m
- Total number of functions = 2^(Number of bits to fiddle with)
- 2^(n*2^(m))



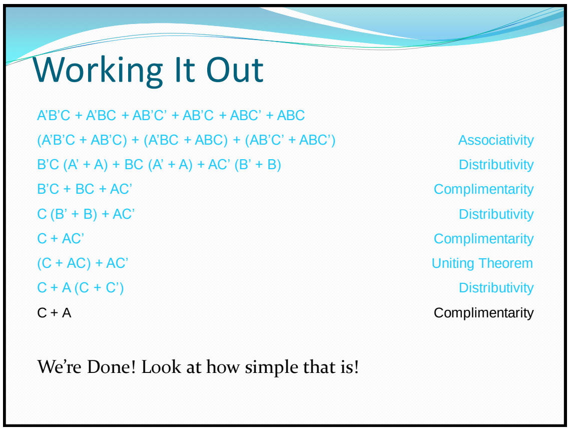
Boolean Equation Minimization

- That equation we just got was gross
  - (A'B'C + A'BC + AB'C' + AB'C + ABC' + ABC)
- There's probably a better way to represent
  - Use Laws of Boolean Algebra
- Also
  - Can help to verify if two functions are the same (they'll minimize to the same thing)
  - Reduces complexity of hardware (most of the time, there are several factors to this…)



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

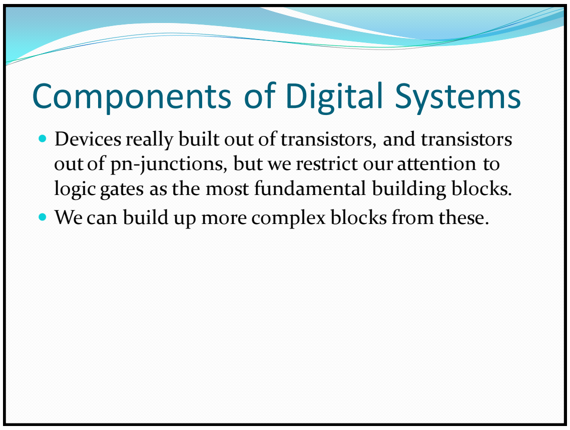


Working It Out

A'B'C + A'BC + AB'C' + AB'C + ABC' + ABC	
(A'B'C + AB'C) + (A'BC + ABC) + (AB'C' + ABC')	Associativity
B'C (A' + A) + BC (A' + A) + AC' (B' + B)	Distributivity
B'C + BC + AC'	Complimentarity
C (B' + B) + AC'	Distributivity
C + AC'	Complimentarity
(C + AC) + AC'	Uniting Theorem
C + A (C + C')	Distributivity
C + A	Complimentarity

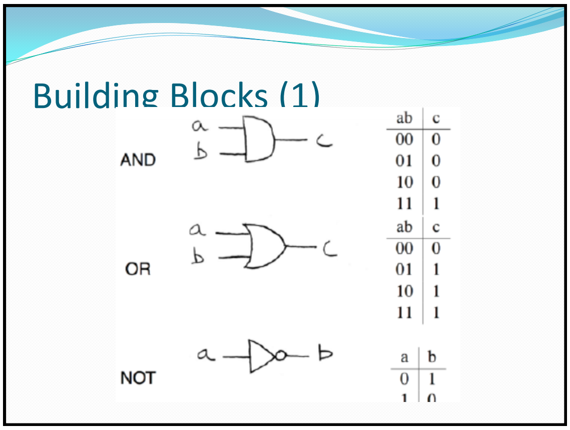
We're Done! Look at how simple that is!

Components of Digital Systems



Components of Digital Systems

- Devices really built out of transistors, and transistors out of pn-junctions, but we restrict our attention to logic gates as the most fundamental building blocks.
- We can build up more complex blocks from these.



Building Blocks (1)

AND

ab	c
00	0
01	0
10	0
11	1

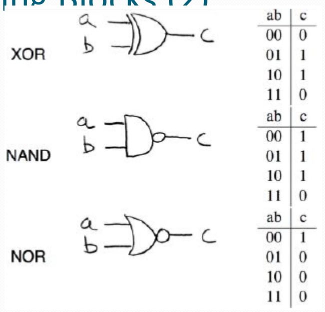
OR

ab	c
00	0
01	1
10	1
11	1

NOT

a	b
0	1
1	0

Building Blocks (2)

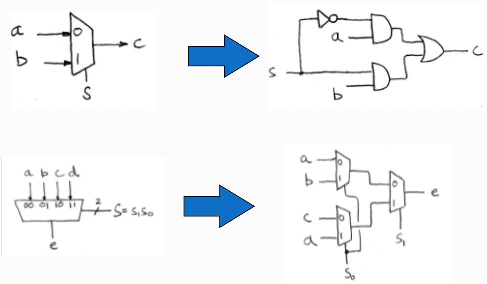


ab	c
00	0
01	1
10	1
11	0

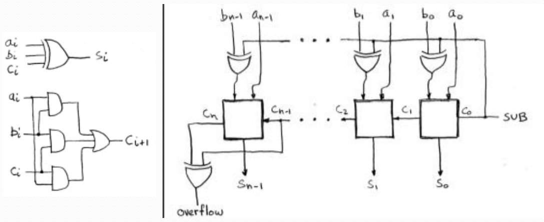
ab	c
00	1
01	1
10	1
11	0

ab	c
00	1
01	0
10	0
11	0

Muxes



Adder



Question S2

- You have 1 of each: mux, OR, NOT.
- How can you represent:
  - Blah = A * C' + B * C' + A' * B' * C
- With only these components?

Answer S2

- Rearrange the Equation:
  - C' * (A + B) + A' * B' * C
- A+B – use the OR
- A' * B' – use NOT on A+B
  - DeMorgan's Law
- Last step is tricky
  - Use each of last two as inputs to the mux, with C as the selection bit

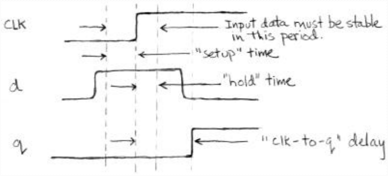
Timing Diagrams

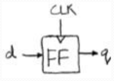
Timing Diagrams

- Why?
  - All real components have delays associated with them
    - Help to show causality in a circuit
    - Make sure you meet timing constraints
  - Different parts of complex systems need to “handshake” somehow
    - Timing Diagrams VERY useful for showing the intricacies of such handshakes
  - They’re what you get when you have to debug a circuit

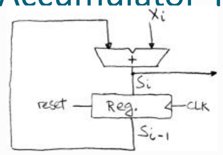
Register Timing

- When cascading registers and combinational logic, must respect setup and hold times.
- Setup:
  - T(clk-to-q)+T(CL)+T(setup) < T(clock)
- Hold:
  - T(clk-to-q) + T(CL) > T(hold)
  - If T(clk-to-q) > T(hold) then don’t need to worry

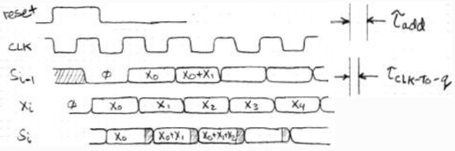




Accumulator Timing



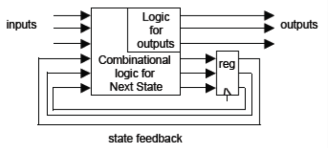
- reset signal shown.
- Also, in practice X might not arrive to the adder at the same time as S_{i-1}
- S_i temporarily is wrong, but register always captures correct value.
- In good circuits, instability never happens around rising edge of clk.



Finite State Machines

Finite State Machines

- Combinational Logic + State Elements
- State Elements essentially keep track of what has been seen so far
- Combinational Logic used to determine what the next state and current output are

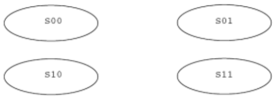


SDS on Fa05 Final (a)

We are designing a circuit with a 1-bit input (I(t)) and a 2-bit output (O(t)), that will produce, at time t, the number of *zeros* in the set {I(t-2), I(t-1), I(t)}. As an example,

```
the input:                 I: 1 1 0 0 1 0 0 1 1 0 1 1 1 0 0 0
...will produce the output: O: 0 0 1 2 2 2 2 2 1 1 1 1 0 1 2 3
```

a) Complete the FSM diagram below. Our states have been labeled Sxy indicating that the previous 2 bits, {I(t-2), I(t-1)} would be {x, y}. Fill in the truth table on the right. The previous state is encoded in (P1,P0), the next state is encoded in (N1,N0), and the output is encoded as (O1,O0). Make sure to indicate the value of the *output* on your state transitions.

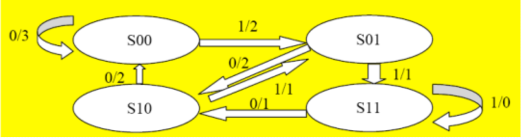


P1	P0	I	O1	O0	N1	N0
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

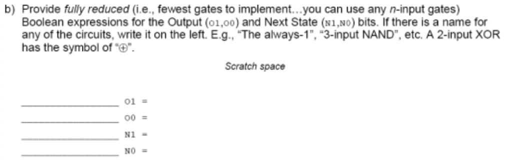
Answer (a)

```
PP I OO NN (Input/Output label for edge) [#ZI(ABC) = NumberOfZerosIn(P1,P0,I)]
10 10 10
--------------
S00 0 -> 11 S00 (0/3) # Had two 0s, another one means we stay here and output
            #ZI(000)=3
S00 1 -> 10 S01 (1/2) # This is our first 1 in a while, register we've seen a 1 by
                      # setting I(t-1) to 1 (i.e., S01) and output #ZI(001)=2
S01 0 -> 10 S10 (0/2) # Saw a 01 before but this 0 means we goto S10 and output
            #ZI(010)=2
S01 1 -> 01 S11 (1/1) # This is the 2nd 1 in a row, go to S11 and output #ZI(011)=1
S10 0 -> 10 S00 (0/2) # Saw a 1 2 timesteps ago, nothing since. Goto S00,output
            #ZI(100)=2
S10 1 -> 01 S01 (1/1) # Saw a 1 2 timesteps ago, a 1 now. Goto 01, output #ZI(101)=1
S11 0 -> 01 S10 (0/1) # Saw 2 straight 1s, now a 0. Goto S10, output #ZI(110)=1
S11 1 -> 00 S11 (1/0) # Everything is coming up 1s! Stay here (in S11), output
                      #ZI(111)=0
```

Answer (a)



SDS on Fa05 Final (b)



b) Provide *fully reduced* (i.e., fewest gates to implement…you can use any *n*-input gates) Boolean expressions for the Output (O1,O0) and Next State (N1,N0) bits. If there is a name for any of the circuits, write it on the left. E.g., "The always-1", "3-input NAND", etc. A 2-input XOR has the symbol of "⊕".

Scratch space

_______________ O1 =

_______________ O0 =

_______________ N1 =

_______________ N0 =

Answer (b)

We'll do the easier ones first. Looking at the truth table (not doing the mindless sum-of-products

calculation), we see:

N0=I

N1=P0

There are no names for these circuits. Let's now look at O1 and O0. If we're extremely clever, we remember the two bit patterns for an adder's two output bits:

O1 is a minority circuit and O0 is a 3-input xnor. Let's see if we can figure that out even if we don't remember these facts. Let's study the truth table and look at the negative spaces (the times when the output is zero). We see when P1 is 0 O0 looks like xnor(P0,I) = ~(P0 XOR I). When P1 is 1 O0 looks like xor(P0,I) = (P0 XOR I). That is, P0 XOR I is being conditionally inverted by P1, which is what an xor does! From this, we see that

O0 = ~[P1 XOR (PO XOR I)], i.e. the post-negation of two cascaded xors, which is the same as a 3-input xnor!

Answer (b)

O1 is a little harder. We can still study the table and see some patterns. That is, when P1 = 0, O1

looks like nand(P0,I) = ~(P0*I). When P1=1, O1 is like a nor(P0,I) = ~(P0+I). This yields

O1 = P1'*(P0*I)' + P1*(P0+I)'

= P1'*(PO'+I') + P1'*(P0'*I') # DeMorgan's law

= P1' P0' + P1' I' + P1' P0' I' # distribution

Recall the following distributive+law-of-1s+identity simplification?
A+AB = A(1+B) = A(1) = A
Well, we can run it backwards. That is, we can start with A and generate A+AB.

We do that here with ~PI~P0:
P1' P0' = P1' P0'(1) = P1' P0'(1+I') = P1' P0' + P1' P0' I'
So that means our three terms for O1 are now four:

Answer (b)

O1 = P1' P0' + P1' I' + P1 P0' I' # from above

O1 = P1' P0' + P1' I' + P1 P0' I' + P1' P0' I' # distributive+law-of-1s+identity

O1 = P1' P0' + P1' I' + (P1+P1')P0' I' # distribution

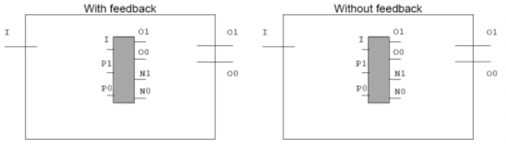
O1 = P1' P0' + P1' I' + (1)P0' I' # complementarity

O1 = P1' P0' + P1' I' + P0' I' # identity

O1 = (P1P0 + P1I + P0I)' # lots more Boolean algebra!
…a NotMajority, or AntiMajority, or Minority circuit!

SDS on Fa05 Final (c)

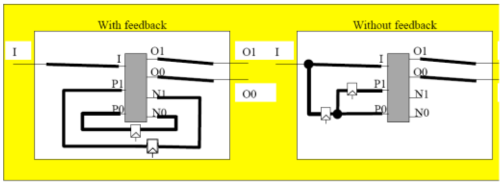
c) Draw the overall circuit using the *fewest* gates possible with and without feedback below. You *may* add registers. “Feedback” means outputs are somehow fed back into inputs. Assume we’ve correctly implemented the answer to (b) as a black box in the middle.



Answer (c)

The feedback circuit is the standard synchronous digital systems model we’ve seen several times, where the output is passed through flip-flops and sent back to the input.

The non-feedback circuit we haven’t seen before. However, from the problem description we know that `sx` and `sy` (i.e., `P1` and `P0`) are really just time-delayed versions of the inputs. I.e., `P0=I(t-1)` and `P1=I(t-2)`, we have the answer on the right.



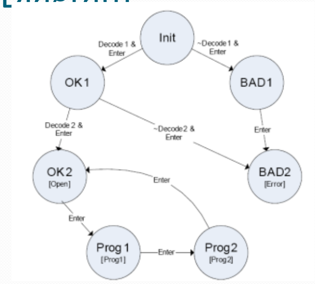
CS150 Lab Problem (1)

- You will be making an **8bit, 2 digit combination lock** such as those sometimes found on secure doors. The inputs to the lock consist of **a code switch** and **two buttons**. The **code switch** is used to enter the digits in the combination. The two buttons are **Reset** which is used to reset the lock and **Enter** which is used to enter a digit of the combination
- The comparison of the current input to each digit will be provided on two wires for you, Decode1 and Decode2

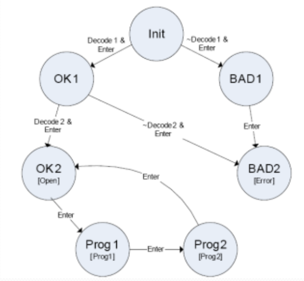
CS150 Lab Problem (2)

- To operate the lock, a user would:
  - 1. **Set the code** to the **first digit** and **press Enter**.
  - 2. **Set the code** to the **second digit** and **press Enter**.
  - 3. The lock will **Open**.
  - 4. The user would then **press enter (SW2)**.
  - 5. **Set the code** to the **new first digit** and **press Enter**.
  - 6. **Set the code** to the **new second digit** and **press Enter**.
  - 7. Cycle back to **step 3** above…
- When someone gets the combo wrong it would go like this:
  - 1. **Set the code** to a **wrong digit** and **press Enter**.
  - 2. **Set the code** to **any digit** (right or wrong) and **press Enter**
  - 3. The lock will show **Error**
  - 4. The lock will stay in this state until the user **presses Reset**.

State Diagram



State Diagram



State Encoding

Init = 0	
OK1 = 1	
OK2 = 2	
Prog1 = 3	
Prog2 = 4	
BAD1 = 5	
BAD2 = 6	

Truth Table

Decode1	Decode2	PS2	PS1	PS0	NS2	NS1	NS0	Open	Error
0	X	0	0	0	1	0	1	0	0
1	X	0	0	0	0	0	1	0	0
X	0	0	0	1	1	1	0	0	0
X	1	0	0	1	0	1	0	0	0
X	X	0	1	0	0	1	1	1	0
X	X	0	1	1	1	0	0	0	0
X	X	1	0	0	0	1	0	0	0
X	X	1	0	1	1	1	0	0	0
X	X	1	1	0	1	1	0	0	1
X	X	1	1	1	X	X	X	X	X

Enter is a necessary condition for all state transitions
Reset will always cause NS to be Init

Boolean Expressions

- NS2 = (PS2’ * PS1’ * PS0’ * Decode1’ + PS2’ * PS1’ * PS0 * Decode2’ + PS2’ * PS1 * PS0 + PS2 * PS1’ * PS0 + PS2 * PS1 * PS0’) * Enter + NS2 * Enter’
- NS1 = (PS2’ * PS1’ * PS0 * Decode2’ + PS2’ * PS1’ * PS0 * Decode2 + PS2’ * PS1 * PS0’ + PS2 * PS1’ * PS0’ + PS2 * PS1’ * PS0 + PS2 * PS1 * PS0’) * Enter + NS1 * Enter’
- NS0 = (PS2’ * PS1’ * PS0 * Decode1’ + PS2’ * PS1’ * PS0 * Decode1 + PS2’ * PS1 * PS0’) * Enter + NS0 * Enter’
- Open = PS2’ * PS1 * PS0’
- Error = PS2 * PS1 * PS0’

Single Cycle CPU Design

Brian Nguyen (Thanks to David Jacobs)

Tasks a CPU must do

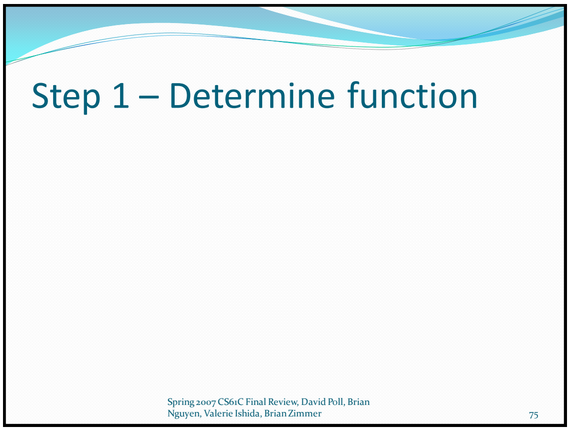
- Fetch an instruction F
- Decode the instruction D
  - Get values from registers and set control lines
- Execute instruction (arithmetic) A
- Meddle with Memory, if necessary M
- Record result of instruction R
  - a.k.a. register write back

Building/Extending a CPU Datapath

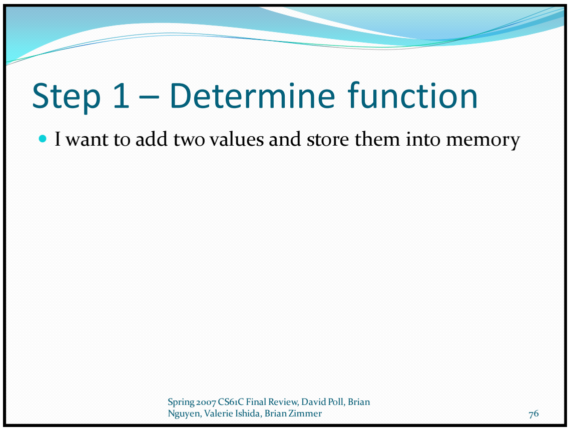
- Determine what function you want to do
  - I want to support adding of two registers
- Determine what you have to work with
  - I have registers, muxes, gates, and lots of wires
- Formulate a plan of bringing data from where it is found to where it is needed ✱
  - I need to move data from registers to an ALU
- Execute your plan
- Determine Control Signals

Applying Those Steps

- If I have the following C code:
-     *p = z + 4;
- Converting it to MIPS would produce
-     addi $t0, z, 4
-     sw $t0, 0(p)
- Let’s suppose you want to do this in 1 instruction

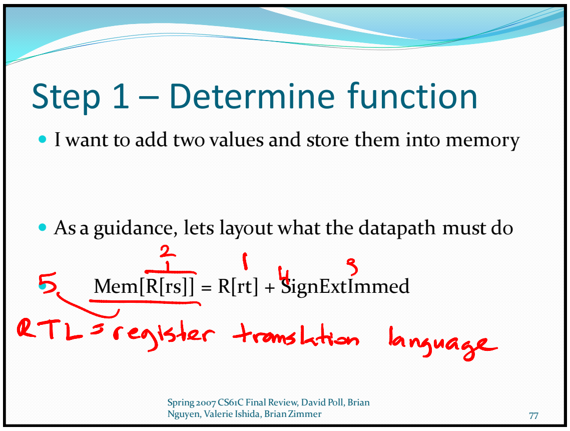


Step 1 – Determine function



Step 1 – Determine function

- I want to add two values and store them into memory



Step 1 – Determine function

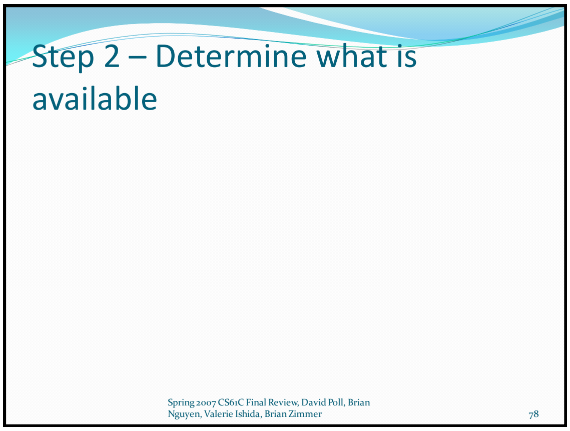
- I want to add two values and store them into memory

- As a guidance, lets layout what the datapath must do

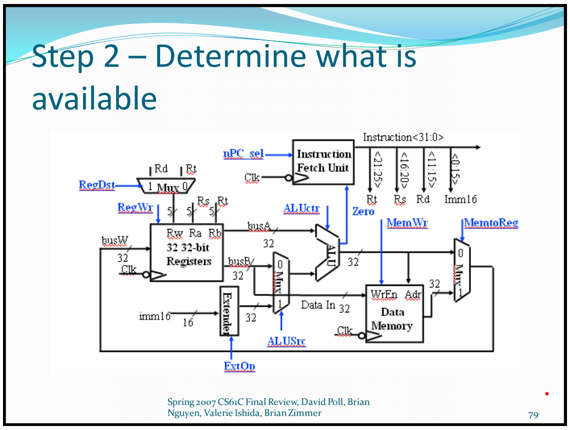
2 1 4 3

5 Mem[R[rs]] = R[rt] + SignExtImmed

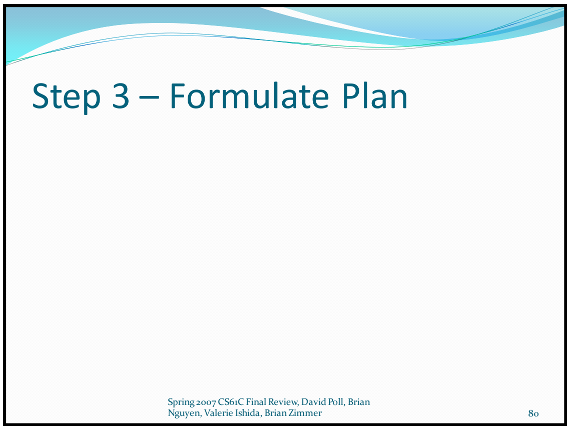
RTL = register translation language



Step 2 – Determine what is available



Step 2 – Determine what is available

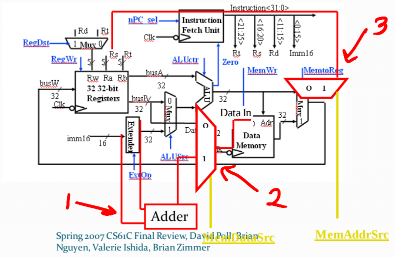


Step 3 – Formulate Plan

Step 3 – Formulate Plan

- Add R[rt] to SignExtImmed
- Send R[rt]+SignExtImmed to Memory Data
- Send R[rs] to Memory Addr

Step 4 – Execute Plan



Step 5 – Set Control Lines

Control	Value	Control	Value
nPC_sel	normal	ExtOp	Sign
RegDst	X	MemWr	1
RegWrite	0	MemToReg	X
ALUCtrl	X	MemDataSrc	1
ALUSrc	X	MemAddrSrc	1

Things to Keep in Mind

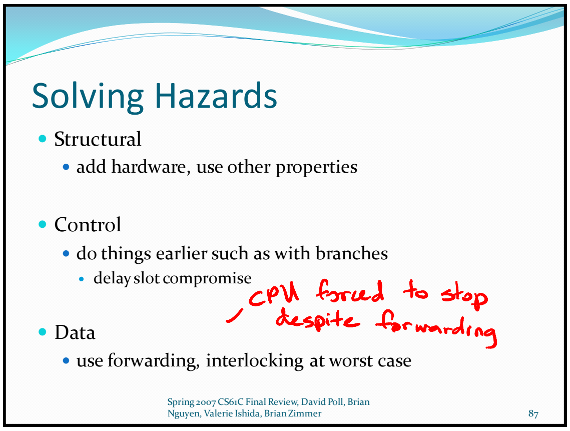
- There is more than one way to modify datapath to produce same result
- If you split a line leading into an input, you need to use a mux.
  - Send original line into 0
  - Send new line into 1

Pipelining

Valerie Ishida (Thanks to Michael Le)

Pipelining Problems

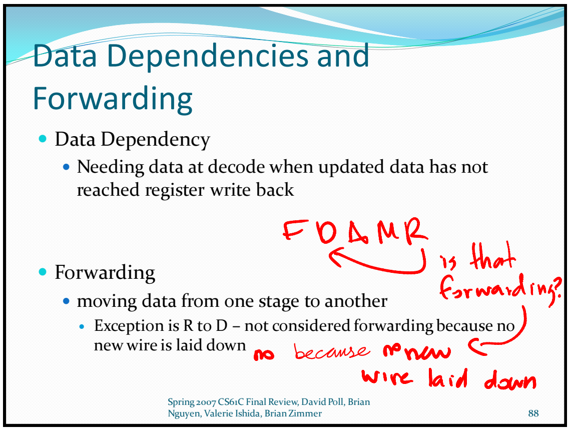
- Hazards
  - Structural: Using some type of circuit two different ways, at the same time
  - Data: Instruction depends on result of prior instruction
  - Control: Later instruction fetches delayed to wait for result of branch



Solving Hazards

- Structural
  - add hardware, use other properties
- Control
  - do things earlier such as with branches
    - delay slot compromise
- Data
  - use forwarding, interlocking at worst case

CPU forced to stop despite forwarding



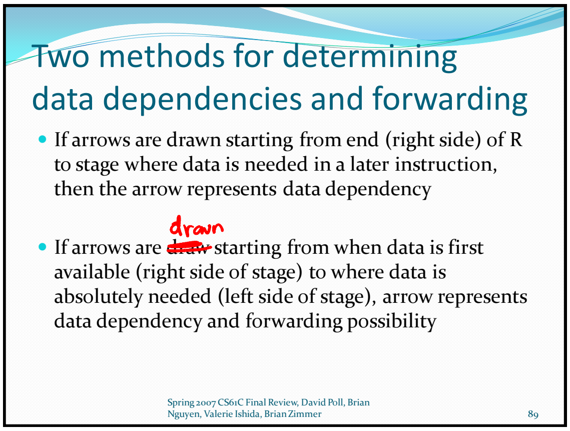
Data Dependencies and Forwarding

- Data Dependency
  - Needing data at decode when updated data has not reached register write back
- Forwarding
  - moving data from one stage to another
    - Exception is R to D – not considered forwarding because no new wire is laid down

F D A M R

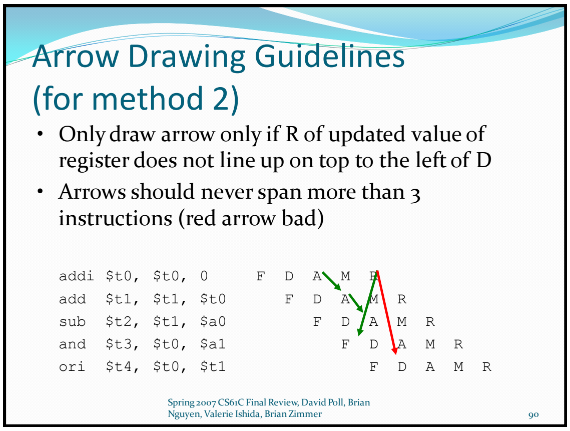
is that forwarding?

no because no new wire laid down



Two methods for determining data dependencies and forwarding

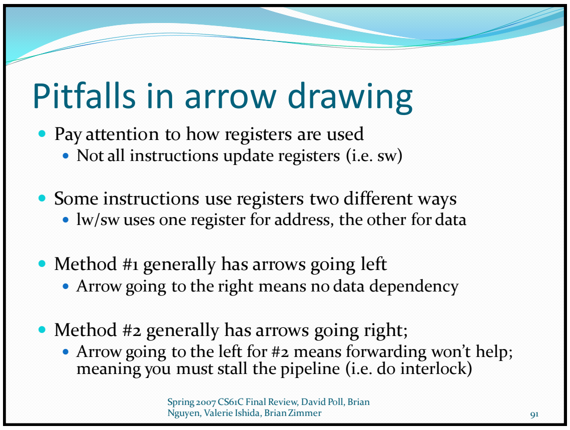
- If arrows are drawn starting from end (right side) of R to stage where data is needed in a later instruction, then the arrow represents data dependency
- If arrows are ~~draw~~ drawn starting from when data is first available (right side of stage) to where data is absolutely needed (left side of stage), arrow represents data dependency and forwarding possibility



Arrow Drawing Guidelines (for method 2)

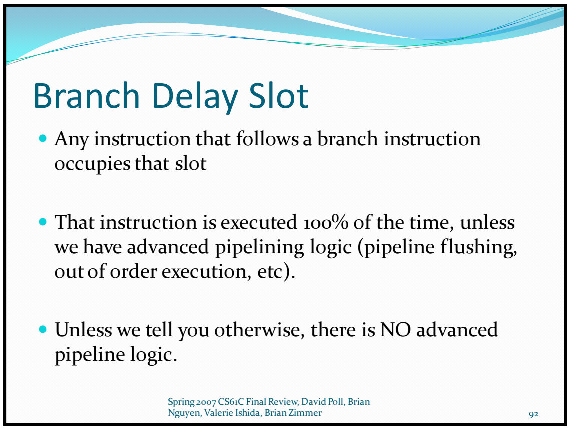
- Only draw arrow only if R of updated value of register does not line up on top to the left of D
- Arrows should never span more than 3 instructions (red arrow bad)

```
addi $t0, $t0, 0     F  D  A  M  R
add  $t1, $t1, $t0      F  D  A  M  R
sub  $t2, $t1, $a0         F  D  A  M  R
and  $t3, $t0, $a1            F  D  A  M  R
ori  $t4, $t0, $t1               F  D  A  M  R
```



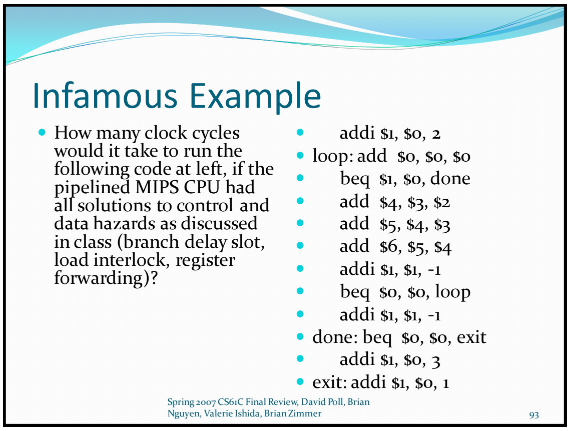
Pitfalls in arrow drawing

- Pay attention to how registers are used
  - Not all instructions update registers (i.e. sw)
- Some instructions use registers two different ways
  - lw/sw uses one register for address, the other for data
- Method #1 generally has arrows going left
  - Arrow going to the right means no data dependency
- Method #2 generally has arrows going right;
  - Arrow going to the left for #2 means forwarding won't help; meaning you must stall the pipeline (i.e. do interlock)



Branch Delay Slot

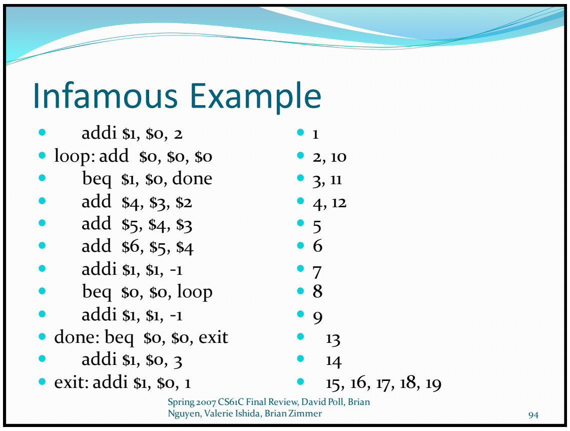
- Any instruction that follows a branch instruction occupies that slot
- That instruction is executed 100% of the time, unless we have advanced pipelining logic (pipeline flushing, out of order execution, etc).
- Unless we tell you otherwise, there is NO advanced pipeline logic.



Infamous Example

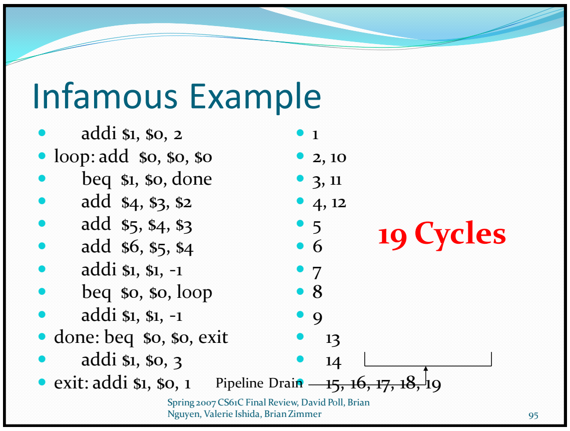
- How many clock cycles would it take to run the following code at left, if the pipelined MIPS CPU had all solutions to control and data hazards as discussed in class (branch delay slot, load interlock, register forwarding)?

```
      addi $1, $0, 2
loop: add  $0, $0, $0
      beq  $1, $0, done
      add  $4, $3, $2
      add  $5, $4, $3
      add  $6, $5, $4
      addi $1, $1, -1
      beq  $0, $0, loop
      addi $1, $1, -1
done: beq  $0, $0, exit
      addi $1, $0, 3
exit: addi $1, $0, 1
```



Infamous Example

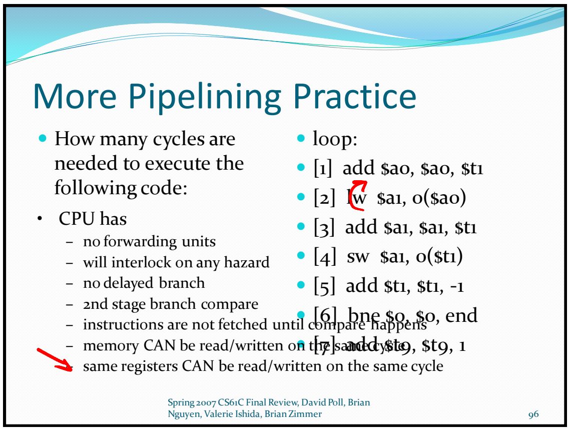
```
      addi $1, $0, 2         1
loop: add  $0, $0, $0        2, 10
      beq  $1, $0, done      3, 11
      add  $4, $3, $2        4, 12
      add  $5, $4, $3        5
      add  $6, $5, $4        6
      addi $1, $1, -1        7
      beq  $0, $0, loop      8
      addi $1, $1, -1        9
done: beq  $0, $0, exit        13
      addi $1, $0, 3           14
exit: addi $1, $0, 1           15, 16, 17, 18, 19
```



Infamous Example

```
      addi $1, $0, 2         1
loop: add  $0, $0, $0        2, 10
      beq  $1, $0, done      3, 11
      add  $4, $3, $2        4, 12
      add  $5, $4, $3        5
      add  $6, $5, $4        6
      addi $1, $1, -1        7
      beq  $0, $0, loop      8
      addi $1, $1, -1        9
done: beq  $0, $0, exit        13
      addi $1, $0, 3           14
exit: addi $1, $0, 1   Pipeline Drain   15, 16, 17, 18, 19
```

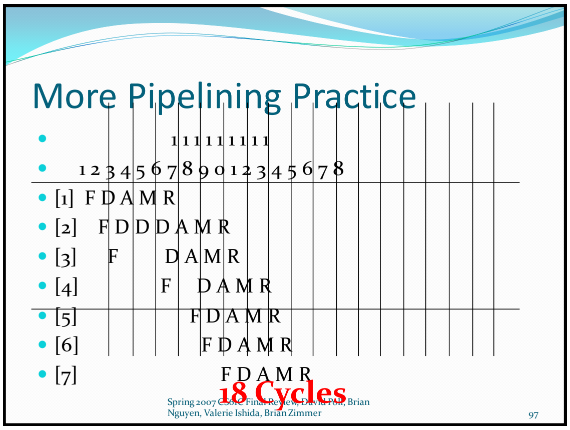
19 Cycles



More Pipelining Practice

- How many cycles are needed to execute the following code:
- CPU has
  - no forwarding units
  - will interlock on any hazard
  - no delayed branch
  - 2nd stage branch compare
  - instructions are not fetched until compare happens
  - memory CAN be read/written on the same cycle
  - same registers CAN be read/written on the same cycle

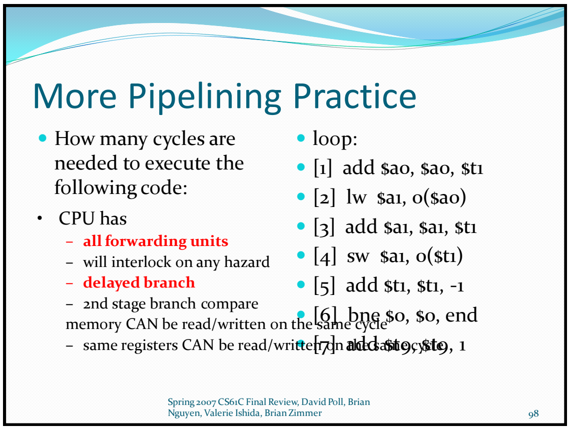
```
loop:
[1]  add $a0, $a0, $t1
[2]  lw  $a1, 0($a0)
[3]  add $a1, $a1, $t1
[4]  sw  $a1, 0($t1)
[5]  add $t1, $t1, -1
[6]  bne $0, $0, end
[7]  add $t9, $t9, 1
```



More Pipelining Practice

```
                    1 1 1 1 1 1 1 1 1
     1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
[1]  F D A M R
[2]    F D D D A M R
[3]      F     D A M R
[4]            F   D A M R
[5]                F D A M R
[6]                  F D A M R
[7]                    F D A M R
```

18 Cycles

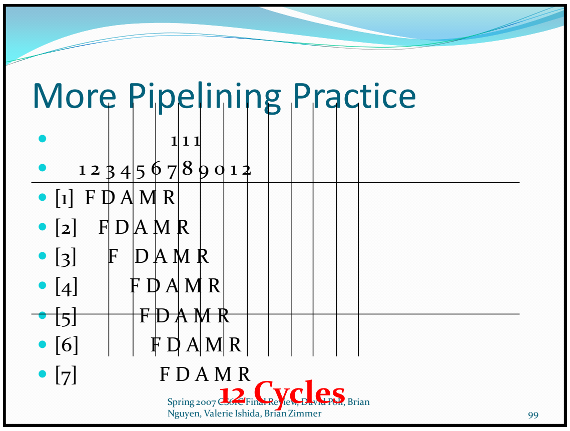


More Pipelining Practice

- How many cycles are needed to execute the following code:
- CPU has
  - **all forwarding units**
  - will interlock on any hazard
  - **delayed branch**
  - 2nd stage branch compare

  memory CAN be read/written on the same cycle
  - same registers CAN be read/written on the same cycle

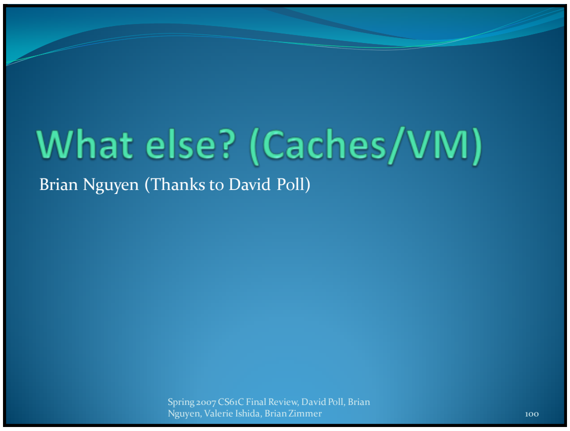
```
loop:
[1]  add $a0, $a0, $t1
[2]  lw  $a1, 0($a0)
[3]  add $a1, $a1, $t1
[4]  sw  $a1, 0($t1)
[5]  add $t1, $t1, -1
[6]  bne $0, $0, end
[7]  add $t9, $t9, 1
```



More Pipelining Practice

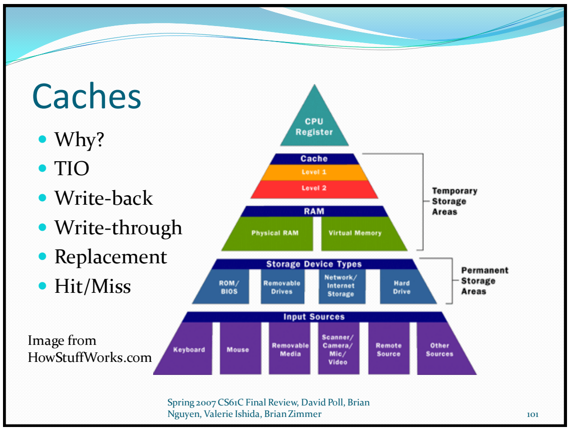
- 1 1 1
- 1 2 3 4 5 6 7 8 9 0 1 2
- [1] F D A M R
- [2] F D A M R
- [3] F D A M R
- [4] F D A M R
- [5] F D A M R
- [6] F D A M R
- [7] F D A M R

12 Cycles



What else? (Caches/VM)

Brian Nguyen (Thanks to David Poll)

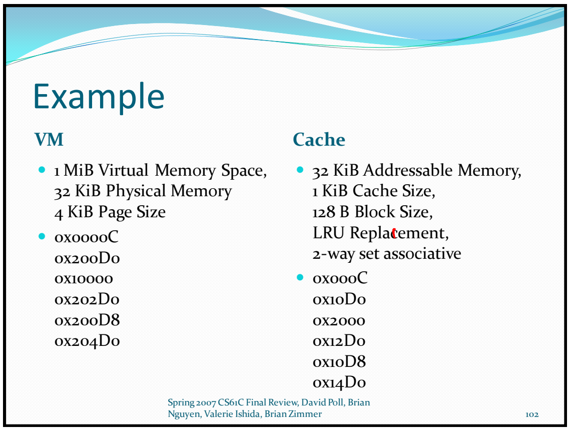


Caches

- Why?
- TIO
- Write-back
- Write-through
- Replacement
- Hit/Miss

CPU Register
Cache
Level 1
Level 2
RAM
Physical RAM
Virtual Memory
Temporary Storage Areas
Storage Device Types
ROM/ BIOS
Removable Drives
Network/ Internet Storage
Hard Drive
Permanent Storage Areas
Input Sources
Keyboard
Mouse
Removable Media
Scanner/ Camera/ Mic/ Video
Remote Source
Other Sources

Image from HowStuffWorks.com



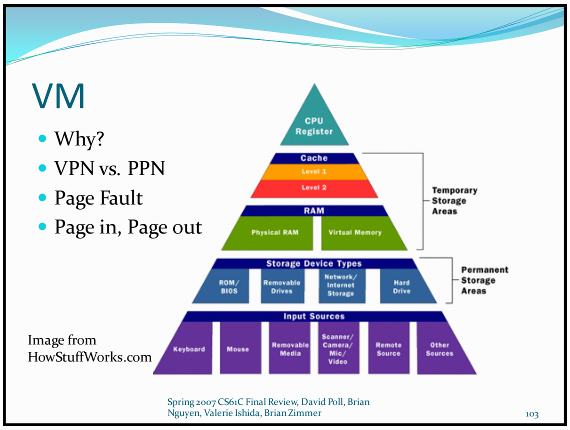
Example

VM

- 1 MiB Virtual Memory Space, 32 KiB Physical Memory 4 KiB Page Size
- 0x0000C
  0x200D0
  0x10000
  0x202D0
  0x200D8
  0x204D0

Cache

- 32 KiB Addressable Memory, 1 KiB Cache Size, 128 B Block Size, LRU Replacement, 2-way set associative
- 0x000C
  0x10D0
  0x2000
  0x12D0
  0x10D8
  0x14D0

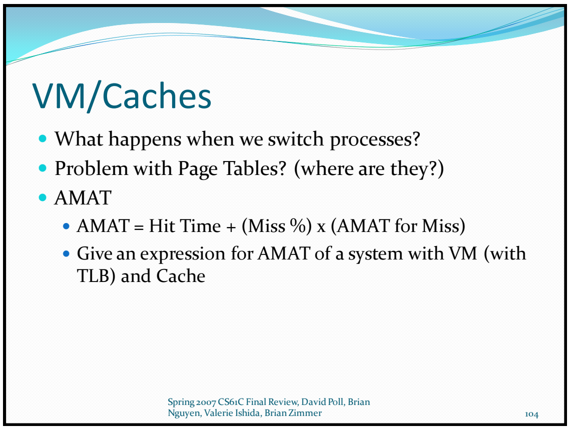


VM

- Why?
- VPN vs. PPN
- Page Fault
- Page in, Page out

CPU Register
Cache
Level 1
Level 2
RAM
Physical RAM
Virtual Memory
Temporary Storage Areas
Storage Device Types
ROM/ BIOS
Removable Drives
Network/ Internet Storage
Hard Drive
Permanent Storage Areas
Input Sources
Keyboard
Mouse
Removable Media
Scanner/ Camera/ Mic/ Video
Remote Source
Other Sources

Image from HowStuffWorks.com



VM/Caches

- What happens when we switch processes?
- Problem with Page Tables? (where are they?)
- AMAT
  - AMAT = Hit Time + (Miss %) x (AMAT for Miss)
  - Give an expression for AMAT of a system with VM (with TLB) and Cache

What else? (Final Potpourri)

Valerie Ishida (Thanks to David Poll)

Performance

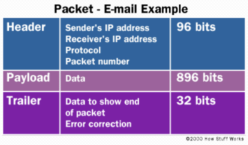
- CPU Time (CPI)
- Example:
  - Memory Read – 10%, CPI = 18
  - Memory Write – 15%, CPI = 20
  - ALU – 30%, CPI = 1
  - Branch – 45%, CPI = 2
  - Overall CPI?
  - CPU Speed = 1 GHz, 1 Million instructions, CPU Time?
  - Cache added. Memory Read/Write halved. Improvement?
- Megahertz Myth
  - What determines performance?

I/O

- Polling
  - Are we there yet?
- Interrupts
  - Wake me when we get there.
- Memory Mapped I/O

Networks

- Sharing vs. Switching
- Half-duplex vs. Full-duplex
- Packets
  - Header
  - Payload
  - Trailer
- Ack?
- TCP/IP



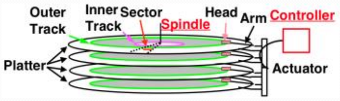
Packet - E-mail Example

Header	Sender's IP address Receiver's IP address Protocol Packet number	96 bits
Payload	Data	896 bits
Trailer	Data to show end of packet Error correction	32 bits

©2000 How Stuff Works

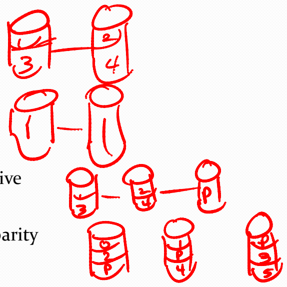
Disks

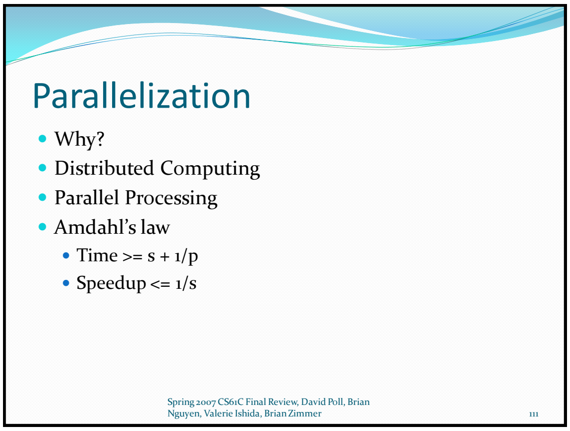
- Latency:
  - Seek Time + Rotation Time + Transfer Time + Controller Overhead



RAID

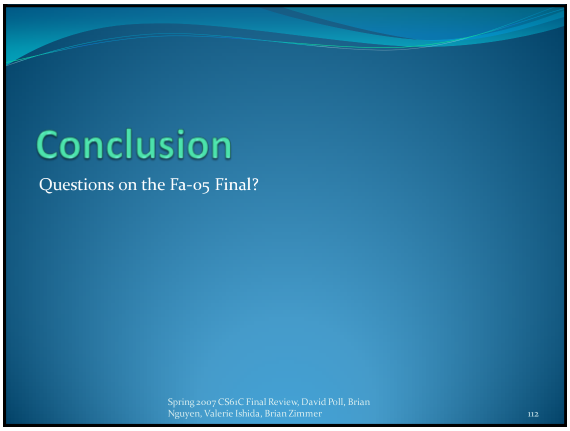
- RAID-0
  - Striped
- RAID-1
  - Mirrored
- RAID-4
  - Striped, parity drive
- RAID-5
  - Striped, striped parity





Parallelization

- Why?
- Distributed Computing
- Parallel Processing
- Amdahl’s law
  - Time >= s + 1/p
  - Speedup <= 1/s



Conclusion

Questions on the Fa-05 Final?