

CS61C Final Review

David Poll, David Jacobs, Michael Le

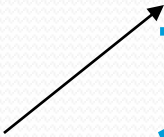
“What’s with all these 1s and 0s?”

David Jacobs

1001	0010	0000	1000
1111	1111	1111	1111

They're a two's complement integer!

“What's with all these 1s and 0s?”

It's negative!  1001 0010 0000 1000
1111 1111 1111 1111

Invert bits and add 1

0110 1101 1111 0111

$(-1) \times (6 \times 16^7 + 11 \times 16^6 + 15 \times 16^5 + 7 \times 16^4 +$

0000 0000 0000 0001

$0 \times 16^3 + 0 \times 16^2 + 0 \times 16^1 + 1 \times 16^0)$

$= -1811349505$
Fall 2006 CS61C Final Review, David Poll, David
Jacobs, Michael Le

They're a floating point number!

“What’s with all these 1s and 0s?”

Sign Exponent Fraction/Significand

• 1 00100100 000100011111111111111111

if denorm

$(-1)^1 \times 1.0001000111...b \times 2^{(36-127)}$

$= -4.323 \times 10^{(-28)}$

Expressed in binary

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

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 ~~\$s0~~, -1(~~\$t0~~)

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

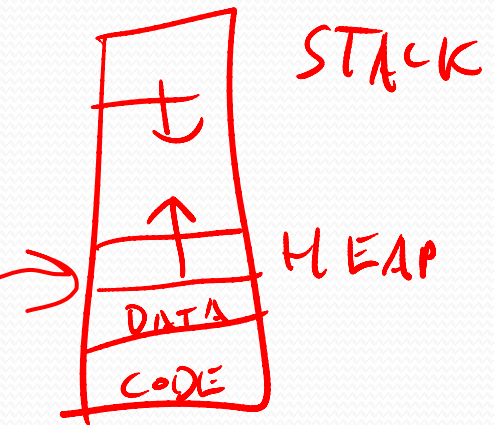


C and Memory

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;
```

```
stack push(stack s, int val){  
    // reserve space  
    // 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!

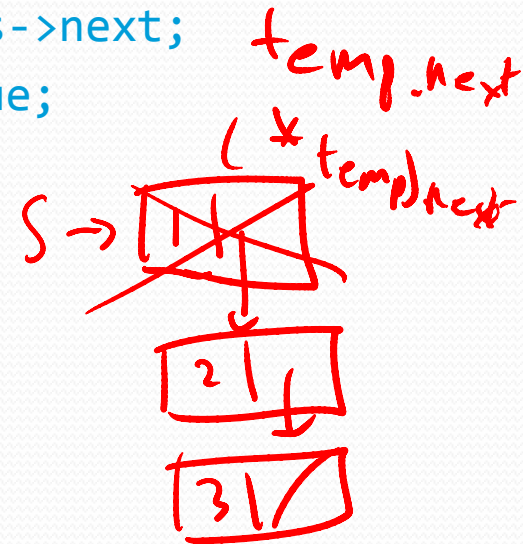
```
typedef struct node {  
    int value;  
    struct node* next;  
} ent;
```

```
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;  
}
```

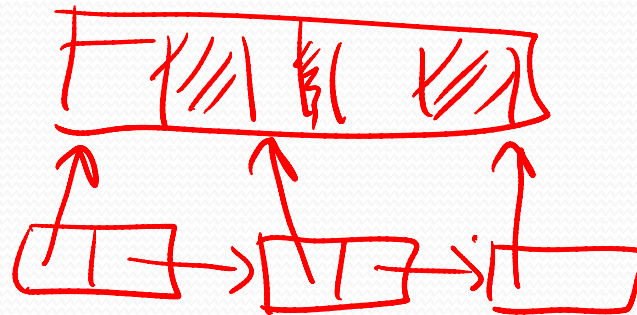
```
stack pop(stack s, int * val){  
    ent * temp = s->next;  
    *val = s->value;  
    free(s);  
    return temp;  
}
```

ent temp



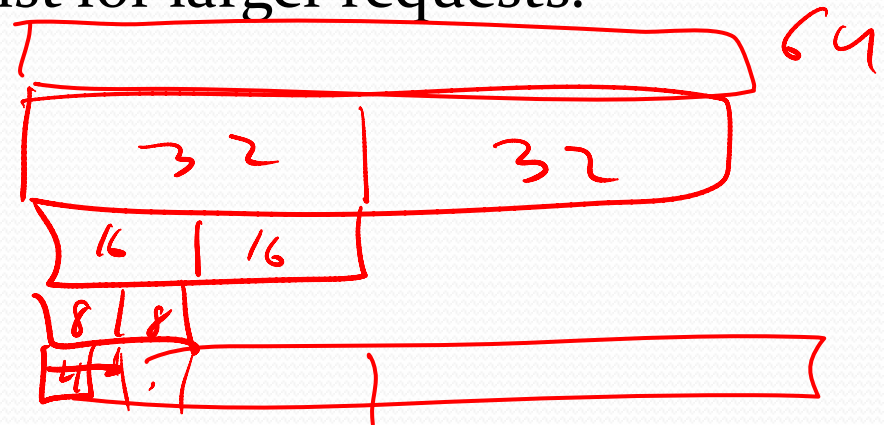
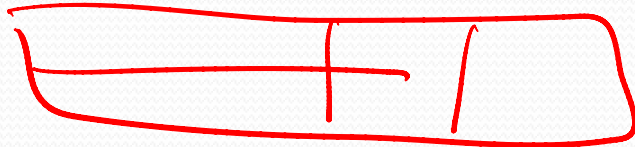
Memory Management

- 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



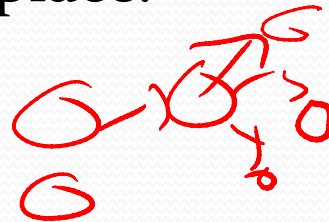
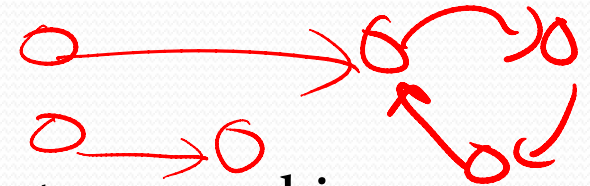
Memory Management

- 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)
```

Saved registers

```
add $s0, $a0, $0
```

Argument registers

Body

```
addiu $a0, $a0, -1
```

```
jal Sum
```

```
add $v0, $v0, $s0
```

Return value

```
lw $s0, 4($sp)
```

Epilogue

```
lw $ra, 0($sp)
```

```
addiu $sp, $sp, 8
```

Return address

```
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;  
}
```



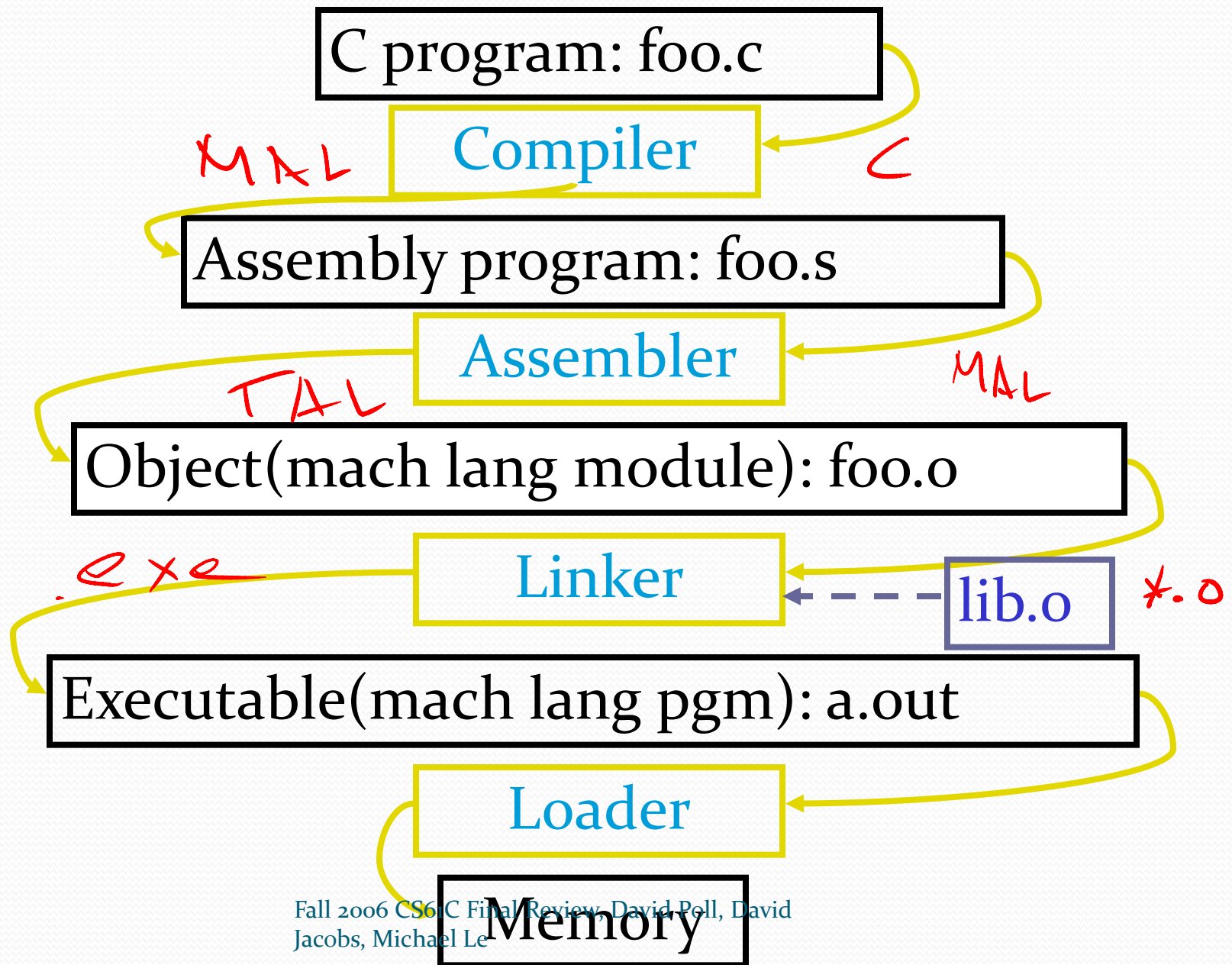
Push:

```
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
```

Handwritten notes in red ink:

char 16 byte
4
5

CALL



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

Michael Le

Problem

You have been hired to build a plate spinning controller for a robot. The robot can only handle the following orientations of a plate:

lean left, balanced, lean right, broken

In addition, there is a wind factor:

strong left, left, right, strong right

Depending on the situation, the robot will respond by pushing the plate **left** or **right**, **spin** the plate, or do **nothing**.

How would you begin designing this circuit?

Finite State Machine

A general approach to designing one

1. Identify the states

orientation

2. Identify the inputs

wind, curr orientation

3. Identify the outputs

new orientation, action

4. Identify the transitions

run through every scenario

Finite State Machine

A general approach to designing one

1. Identify the states

Orientation

2. Identify the inputs

Wind, Current State

3. Identify the outputs

Action, Next State

4. Identify the transitions

**Find all
possible combos**

Additional Problem Information

The robot will respond as follows:

- If the wind is strong, the orientation shifts two steps.
 - This means, **Left** + **Strong Left** = **Broken Plate**
- Robot spins plate only when plate returns to the **balanced** state due to the wind
- Once plate is **broken**, controller does nothing

What does the FSM look like?

Inputs

- CurrentState
 - **Left, Right, Balance, Broken**
- Wind
 - **Strong Left, Left, Right, Strong Right**

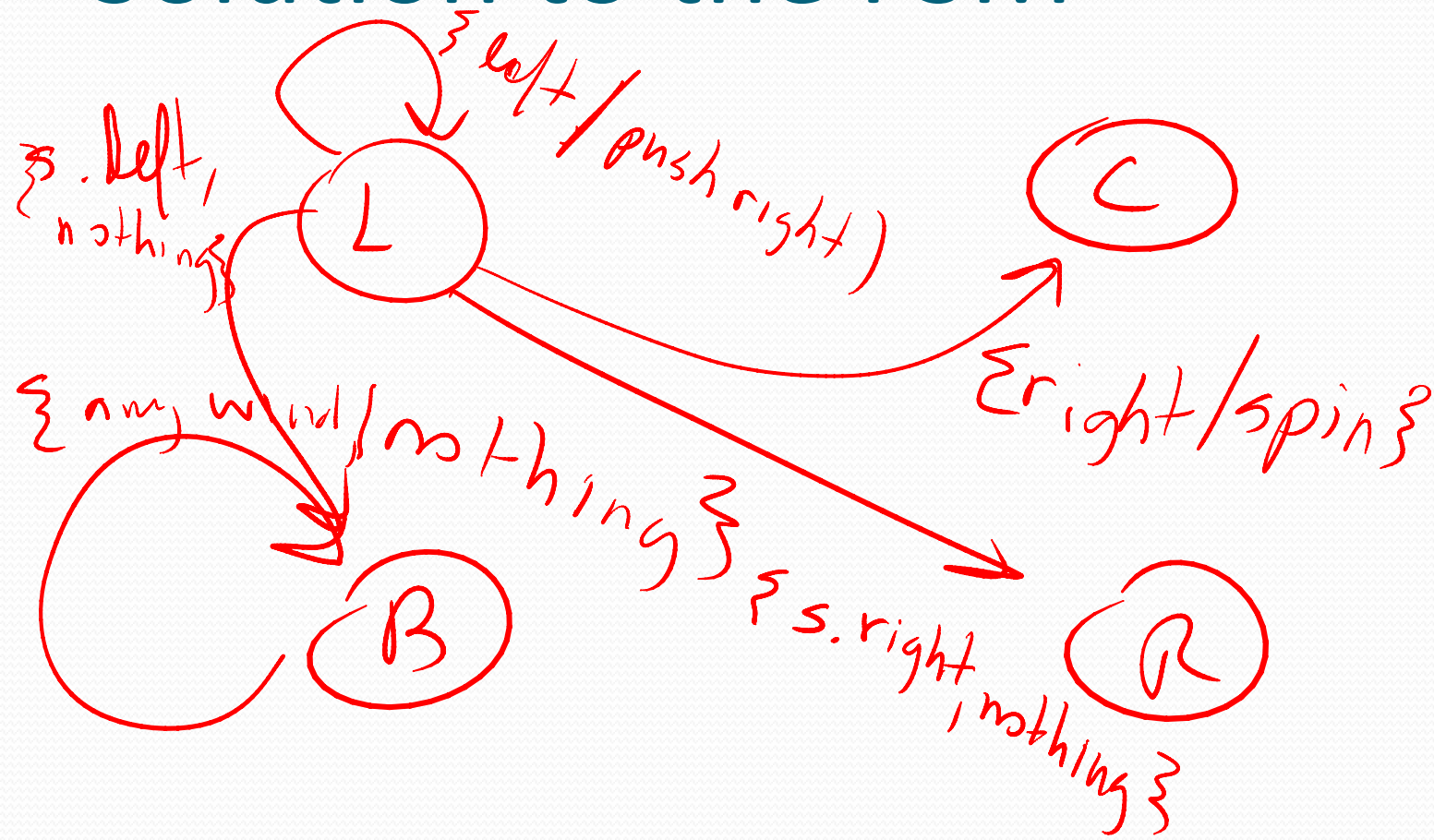
Outputs

- NextState
 - **Left, Right, Balance, Broken**
- Action
 - **Push left, spin, push right, do nothing**

The robot will respond as follows:

- If the wind is strong, the orientation shifts two steps.
 - This means, **Left** + **Strong Left** is a **Broken Plate**
- Robot spins plate only when plate returns to the **balanced** state due to the wind
- Once plate is **broken**, controller does nothing

Partial ✓ Solution to the FSM



What Next?

- Now that we have an FSM, what do we do now?

Building Truth Tables

Two general methods

- Running through every combination of the inputs
- If an input/output is multiple bits, break treat each bit as an individual input
- Follow all the transition arcs of your FSM

Solution to the Truth Table

State

10 - L

11 - C

01 - R

00 - B

Wind

10 - S, L

11 - S, R

00 - L

01 - R

Curr1	Curr0	Wind1	Wind0	Next1	Next0	Out1	Out0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	1	1	1	1
0	1	0	1	1	1	1	0
0	1	1	0	1	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	1	1	0	1
1	0	0	1	1	1	1	1
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	1	0	0	0
1	1	0	1	0	1	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0

Act 1

Act 0

Act

01 - push right

10 - push left

11 - spin

00 - nothing

Going from Truth Table to Circuit

- Canonical Sums of Products
 - For each output, **OR** every combination that produces a true value
 - Each combination depends on **AND**'ed inputs
 - ~~Commonly~~ known as the Brute-Force method
otherwise
- For example, for majority circuit

$$\text{Maj}(A, B, C) = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + A \cdot B \cdot C$$

Develop your Expressions

- Using your truth table, determine the expressions for Next_1 , Next_0 , Act_1 , and Act_0 .

Brute Force Result

$$\text{Next}_1 = \overline{C}_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + \overline{C}_1 \overline{C}_0 \overline{W}_1 W_0 + \overline{C}_1 \overline{C}_0 W_1 \overline{W}_0 + \\ C_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 W_1 \overline{W}_0$$

$$\text{Next}_0 = \overline{C}_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + \overline{C}_1 \overline{C}_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + \\ C_1 \overline{C}_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 W_1 \overline{W}_0 + C_1 \overline{C}_0 W_1 W_0$$

$$\text{Act}_1 = \overline{C}_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + \overline{C}_1 \overline{C}_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 \overline{W}_1 \overline{W}_0$$

$$\text{Act}_0 = \overline{C}_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 W_0$$

Reflecting on Brute Force

- Easy, but ugly.
- Sometimes not the optimal solution
- What can we do to get a more elegant result?

Boolean Algebra: Elegant Solution

Use Boolean Algebra and simplify your expressions!

$$x \cdot \bar{x} = 0$$

$$x \cdot 0 = 0$$

$$x \cdot 1 = x$$

$$x \cdot x = x$$

$$x \cdot y = y \cdot x$$

$$(xy)z = x(yz)$$

$$x(y + z) = xy + xz$$

$$xy + x = x$$

$$\overline{x \cdot y} = \bar{x} + \bar{y}$$

$$x + \bar{x} = 1$$

$$x + 1 = 1$$

$$x + 0 = x$$

$$x + x = x$$

$$x + y = y + x$$

$$(x + y) + z = x + (y + z)$$

$$x + yz = (x + y)(x + z)$$

$$(x + y)x = x$$

$$\overline{(x + y)} = \bar{x} \cdot \bar{y}$$

complementarity

laws of 0's and 1's

identities

idempotent law

commutativity

associativity

distribution

uniting theorem

DeMorgan's Law

Elegant Solution

$$\text{Next}_1 = \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + \overline{C}_1 C_0 \overline{W}_1 W_0 + \overline{C}_1 C_0 W_1 \overline{W}_0 + \\ C_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 W_1 \overline{W}_0$$

$$\text{Next}_0 = \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + \overline{C}_1 C_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + \\ C_1 \overline{C}_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 W_1 \overline{W}_0 + C_1 C_0 \overline{W}_1 W_0$$

Brute Force

Simplified

$$\text{Next}_1 = \overline{C}_1 C_0 \overline{W}_1 + \overline{C}_1 C_0 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 + C_1 \overline{W}_1 \overline{W}_0$$

$$\text{Next}_0 = \overline{C}_1 C_0 \overline{W}_1 + C_0 \overline{W}_1 W_0 + C_1 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 W_0$$

Elegant Solution

$$\text{Act}_1 = \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + \overline{C}_1 C_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 \overline{W}_1 W_0$$

$$\text{Act}_0 = \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 W_0$$

Brute Force

Simplified

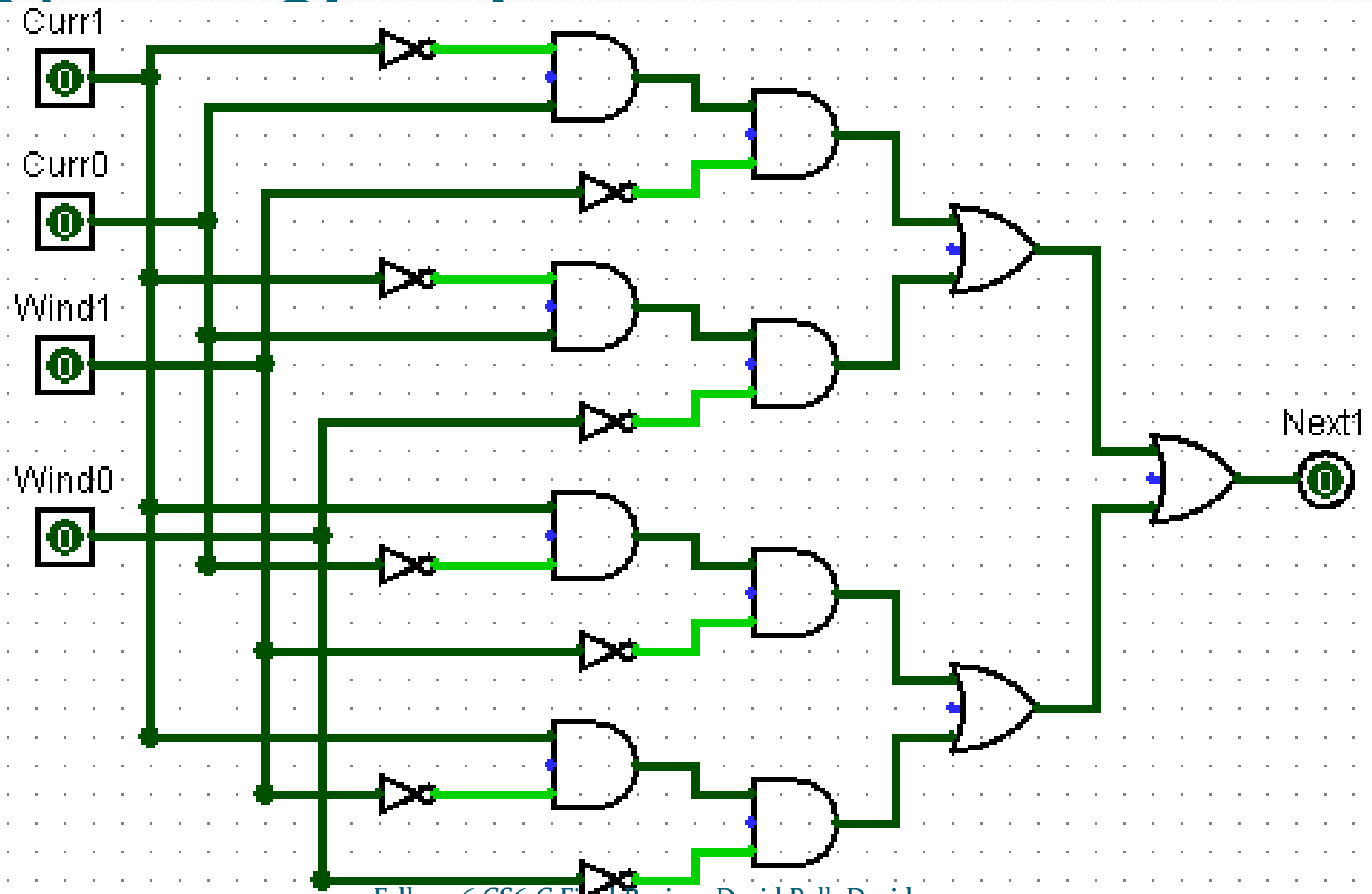
$$\text{Act}_1 = \overline{C}_1 C_0 \overline{W}_1 + C_1 \overline{C}_0 \overline{W}_1 W_0$$

$$\text{Act}_0 = \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1$$

Expressions to Gates

- With your expressions, conversion to gates is mechanical using the sums of products approach
 - Each term becomes an **AND** gate
 - Collect the output of the appropriate **AND** gate into an **OR**

Next₁ circuit

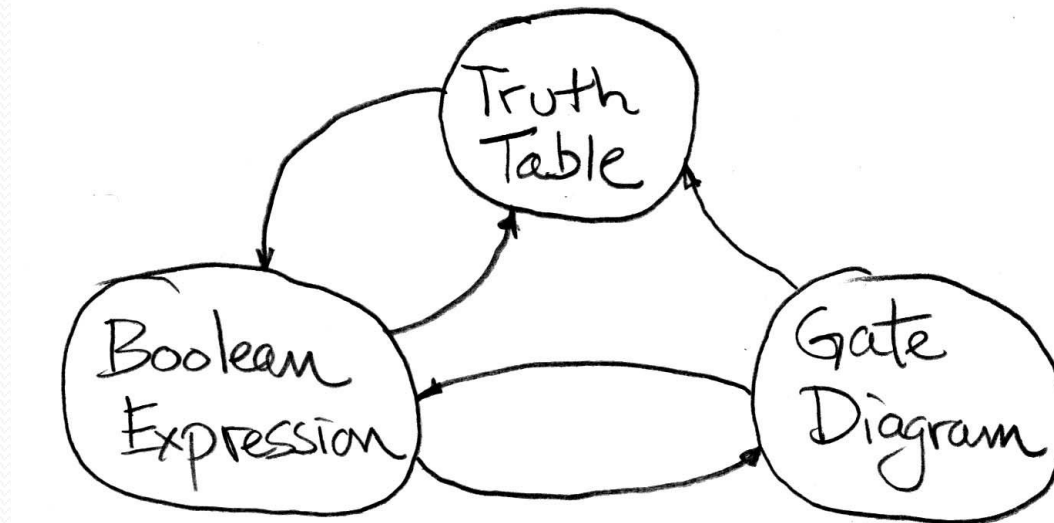


The remaining circuits...

- They are quite trivial and I'm sure you didn't want me to draw them for you
- Don't forget about registers!
 - they are used to hold state

SDS Review

- Master SDS is all about mastering the Trifecta[®]




- It is possible to transition from any state to any other state. However, the ease of this transition is dependent on the complexity of the problem

Single Cycle CPU Design

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

1. Determine what function you want to do
 - I want to support adding of two registers
2. Determine what you have to work with
 - I have registers, muxes, gates, and lots of wires
3. 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
4. Execute your plan
5. 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

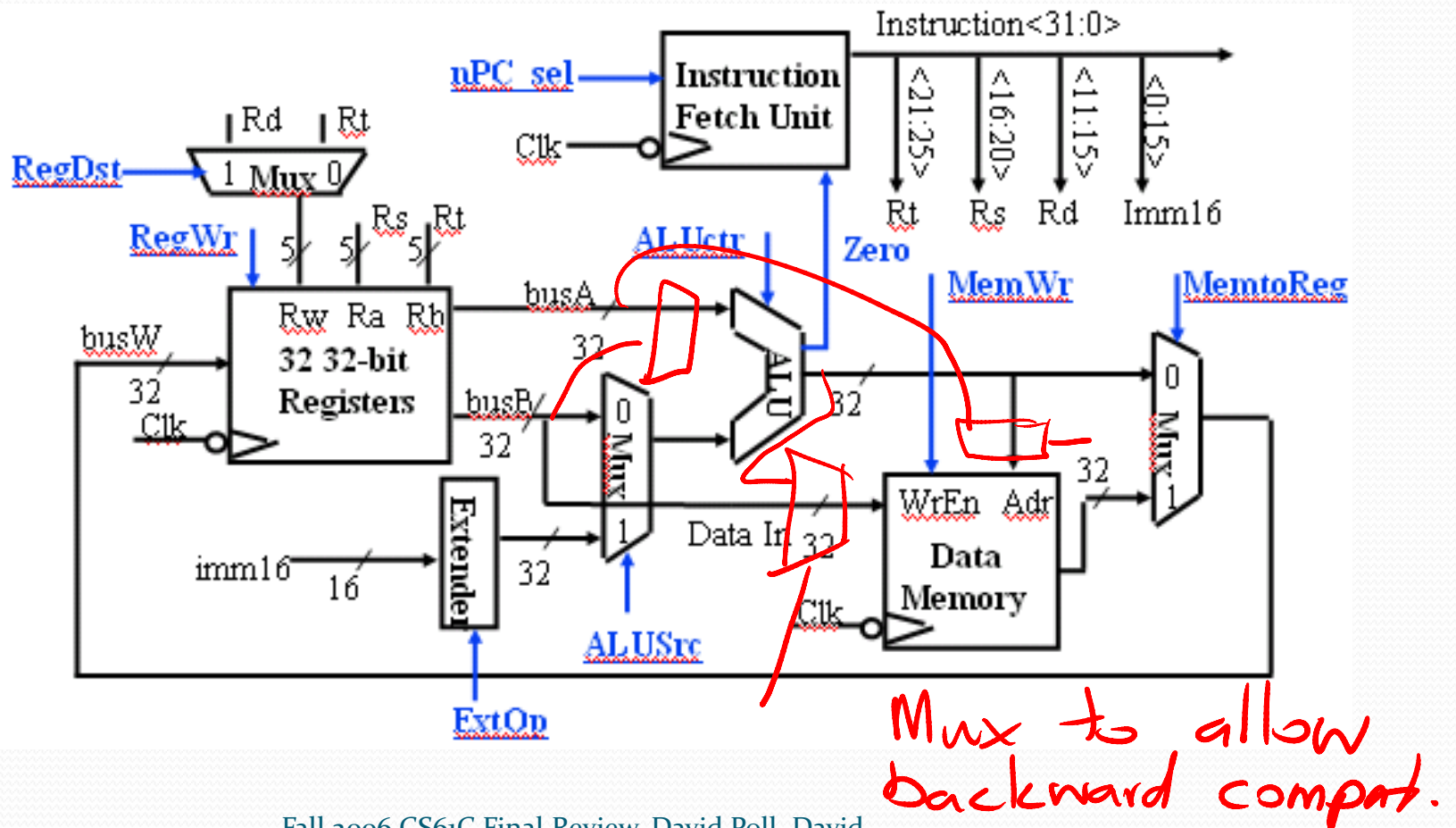
$$\text{Mem}[\text{R}[\text{rs}]] = \text{R}[\text{rt}] + \text{SignExtImm}$$

RTL = register translation language



Step 2 – Determine what is available

Step 2 – Determine what is available



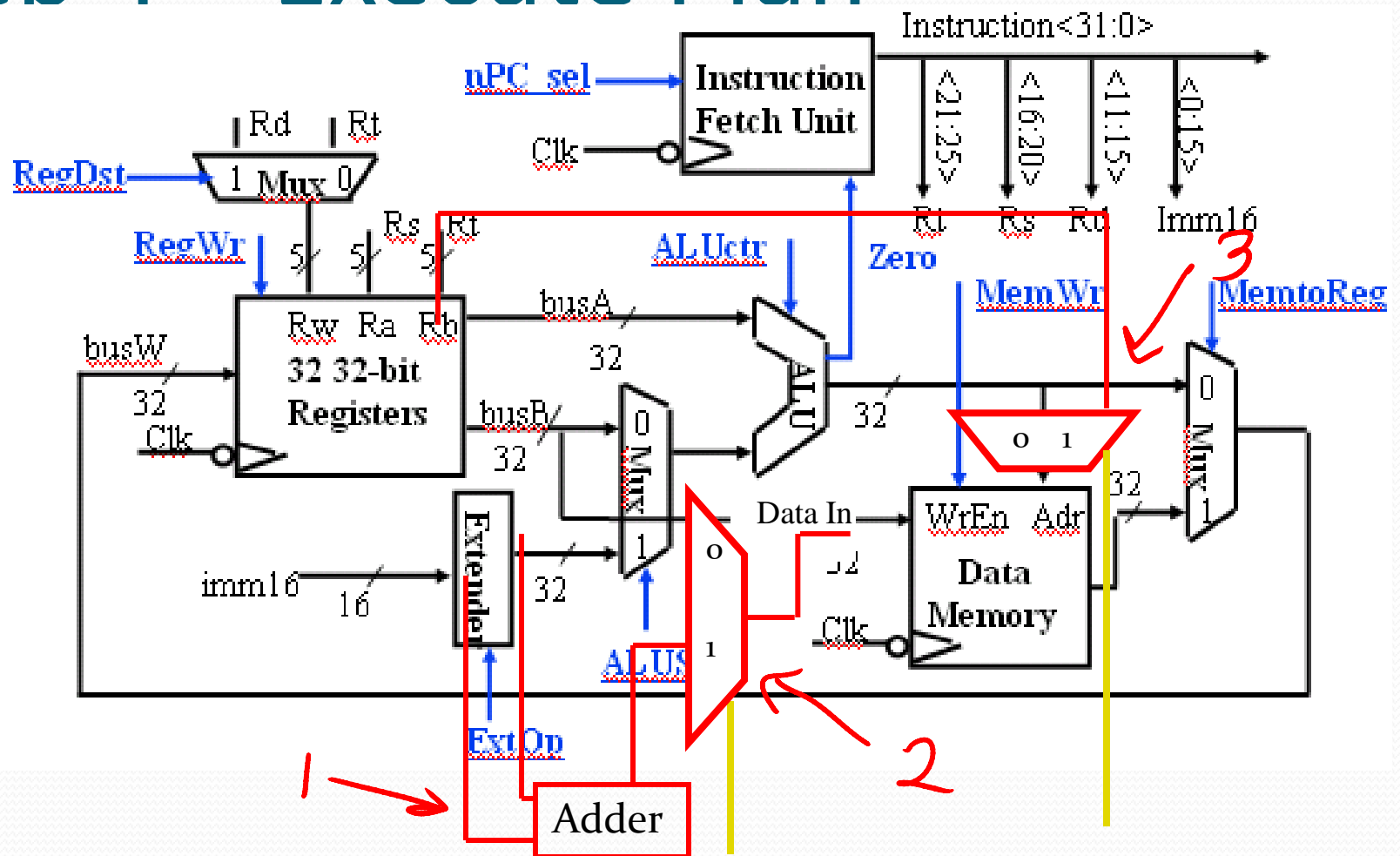


Step 3 – Formulate Plan

Step 3 – Formulate Plan

1. Add $R[rt]$ to SignExtImmed
2. Send $R[rt] + \text{SignExtImmed}$ to Memory Data
3. 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

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

1. 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
2. If arrows are ^{drawn}~~draw~~ 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

The diagram illustrates register update flow across five instructions. Green arrows show valid updates: from \$t0 to \$t1 (row 1 to 2), \$t1 to \$t2 (row 2 to 3), and \$t2 to \$t3 (row 3 to 4). A red arrow shows an invalid update from \$t0 to \$t4 (row 1 to 5), as it spans more than 3 instructions. The columns are labeled F, D, A, M, R, representing different stages of the instruction.

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	15,	16, 17, 18, 19

19 Cycles

Pipeline Drain 

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	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
										1	1	1	1	1	1	1	1	1
										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 comparememory 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	2	3	4	5	6	7	8	9	10	11	12
[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?

David Poll

Caches

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

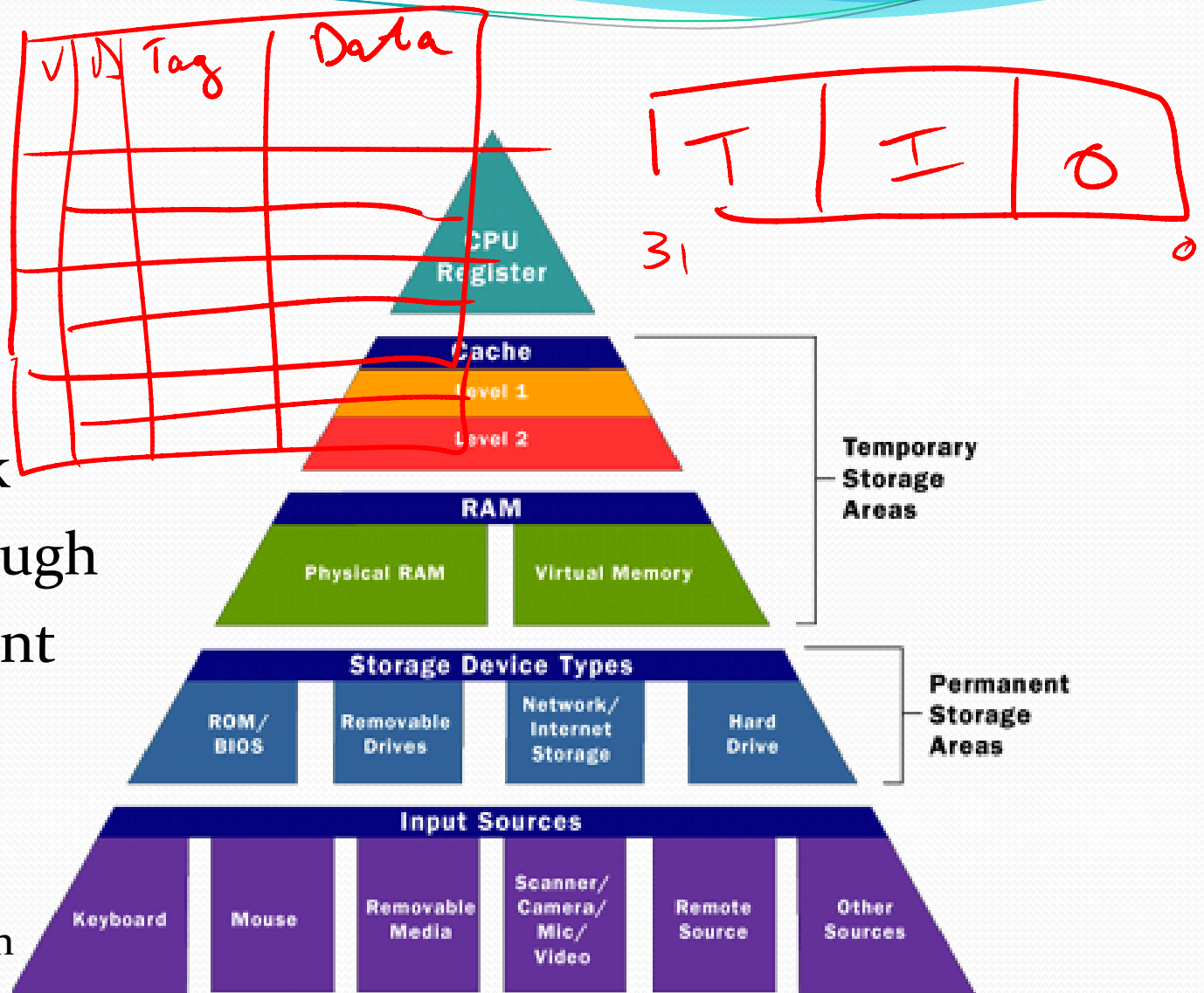


Image from
HowStuffWorks.com

Example

VM

- 1 MiB Virtual Memory Space,
32 KiB Physical Memory
4 KiB Page Size

0x0000C
 0x200D0
 0x10000
 0x202D0
 0x200D8
 0x204D0

VPN | offset

VPN PPN

1) 0 → 0 000C
 20 → 1 10D0
 10 → 2 200B
 12D6

256 VP
8 bit VPN

3 bit PPN

Cache

- 32 KiB Addressable Memory,
1 KiB Cache Size,
128 B Block Size,
LRU Replacement,
2-way set associative

		Index		Data		
		9/15 bits	7/6 bits	offset		
M	0x000C	0	0	000C		
M	0x10D0		0	2000		
M	0x2000		0	10000		
M	0x12D0		0	001000	1080	
H	0x10D8	1	0	001001	1280	
	0x14D0		0			
		2				
		3				

001 0000 1100 0000

VM

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

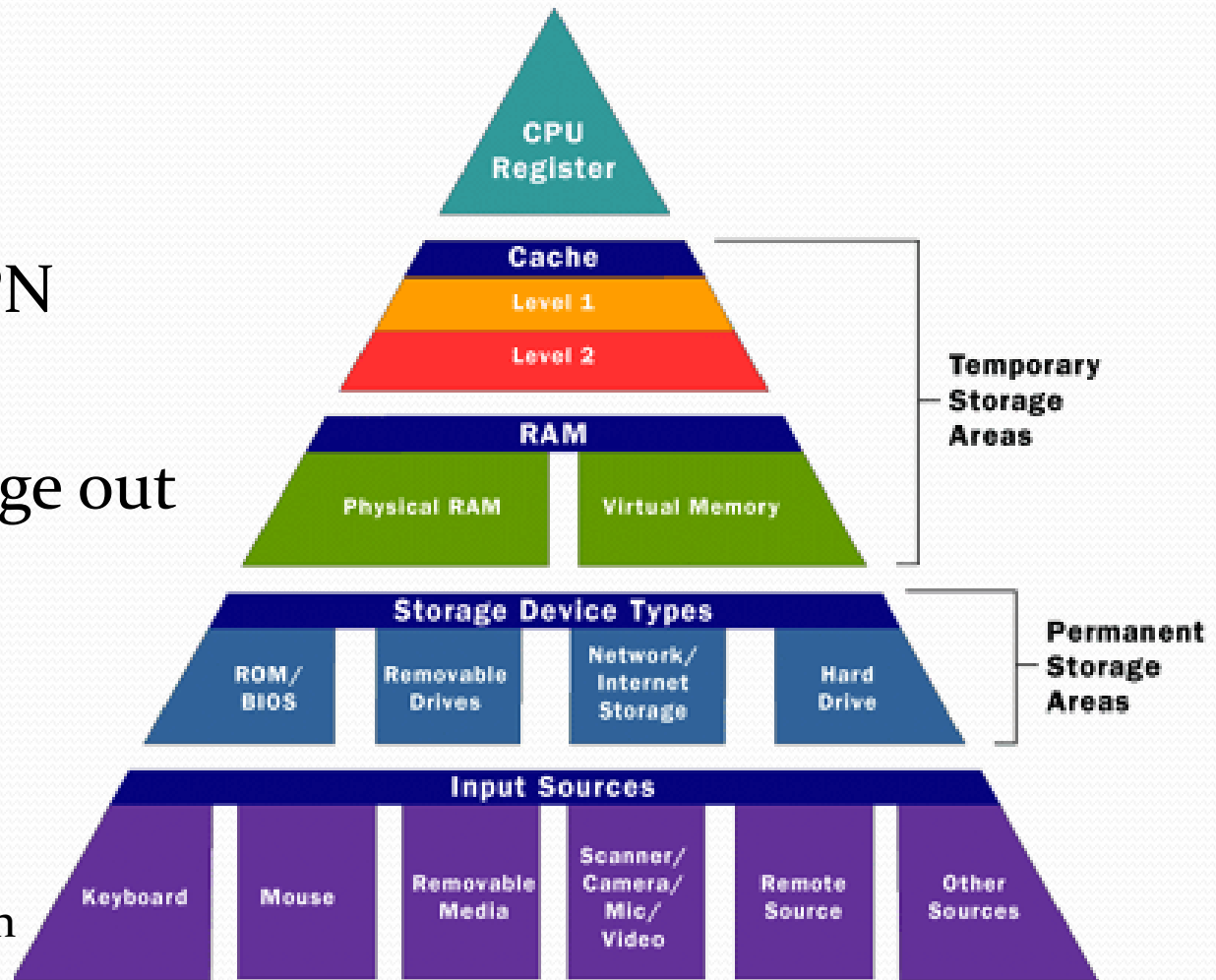


Image from
HowStuffWorks.com

VM/Caches

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

Performance

- CPU Time (CPI)

- Example:

- Memory Read – 10%, CPI = 18 = 1.8
- Memory Write – 15%, CPI = 20 = 3.0
- ALU – 30%, CPI = 1 = .3
- Branch – 45%, CPI = 2 = .9
- Overall CPI?
- CPU Speed = 1 GHz, 1 Million instructions, CPU Time?
- Cache added. Memory Read/Write halved. Improvement?

- Megahertz Myth

- What determines performance?

6ms
CPU Time

6.0 cycles/instruction

$1 \times 10^9 \text{ cycle/s}$

$\frac{6 \times 10^6 \text{ cycles}}{1 \times 10^9} = 6 \times 10^{-3}$

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

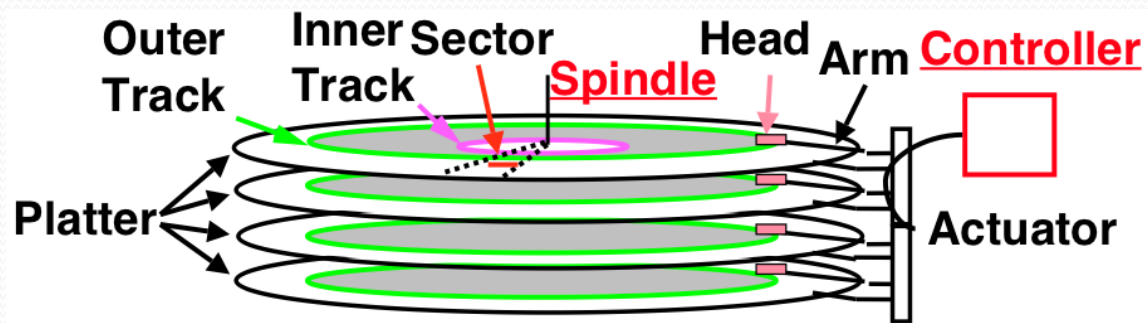
Packet - E-mail Example

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

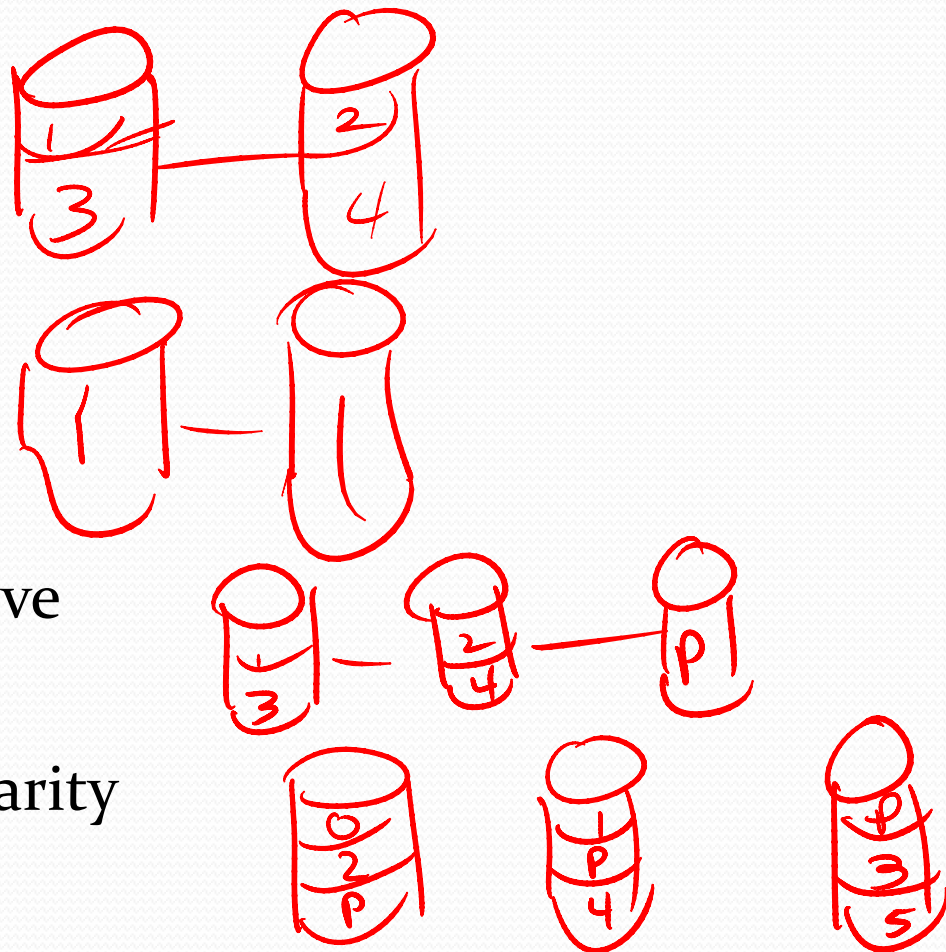
Disks

- 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
 - $\text{Time} \geq s + 1/p$
 - $\text{Speedup} \leq 1/s$

Conclusion

Questions on the Sp-04 Final?