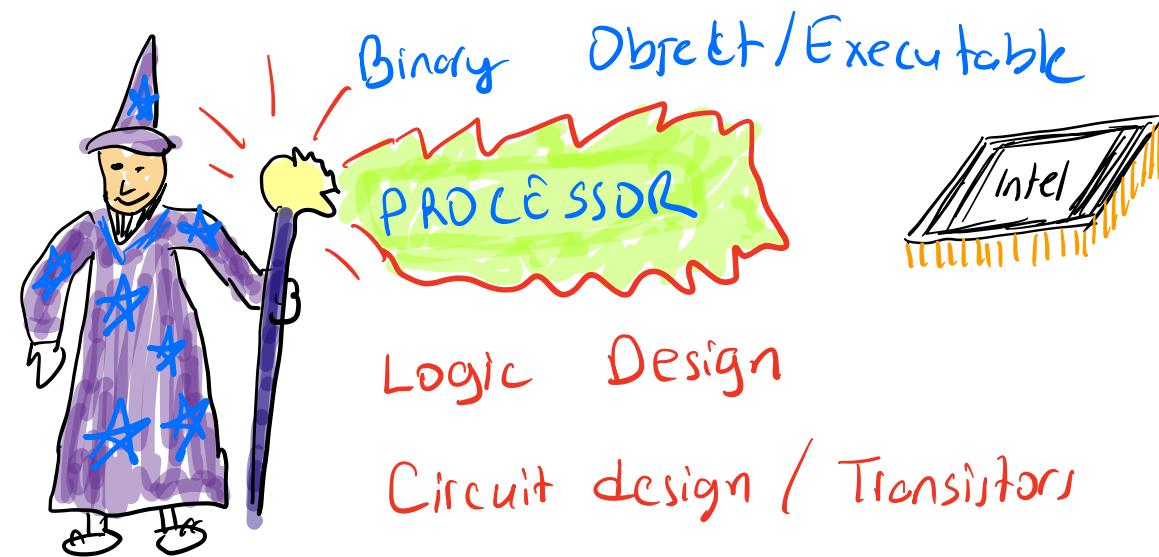


# Processor Architecture: ISA-Y86/MIPS



# Course Outline

— for the next couple of weeks

THIS WEEK

- **Background**
  - Instruction sets
  - Logic design
- **Sequential Implementation**
  - A simple, but not very fast processor design
- **Pipelining**
  - Get more things running simultaneously
- **Pipelined Implementation**
  - Make it work
- **Advanced Topics**
  - Performance analysis
  - High performance processor design

# Coverage

## ■ Our Approach

- Work through designs for particular instruction set
  - Y86-64 – a simplified version of the Intel x86-64
  - If you know one, you more-or-less know them all
- Work at “microarchitectural” level
  - Assemble basic hardware blocks into overall processor structure
    - Memories, functional units, etc.
  - Surround by control logic to make sure each instruction flows through properly
- Use simple hardware description language to describe control logic
  - Can extend and modify
  - Test via simulation
  - Route to design using Verilog Hardware Description Language
    - See Web aside ARCH:VLOG

~~ISA~~

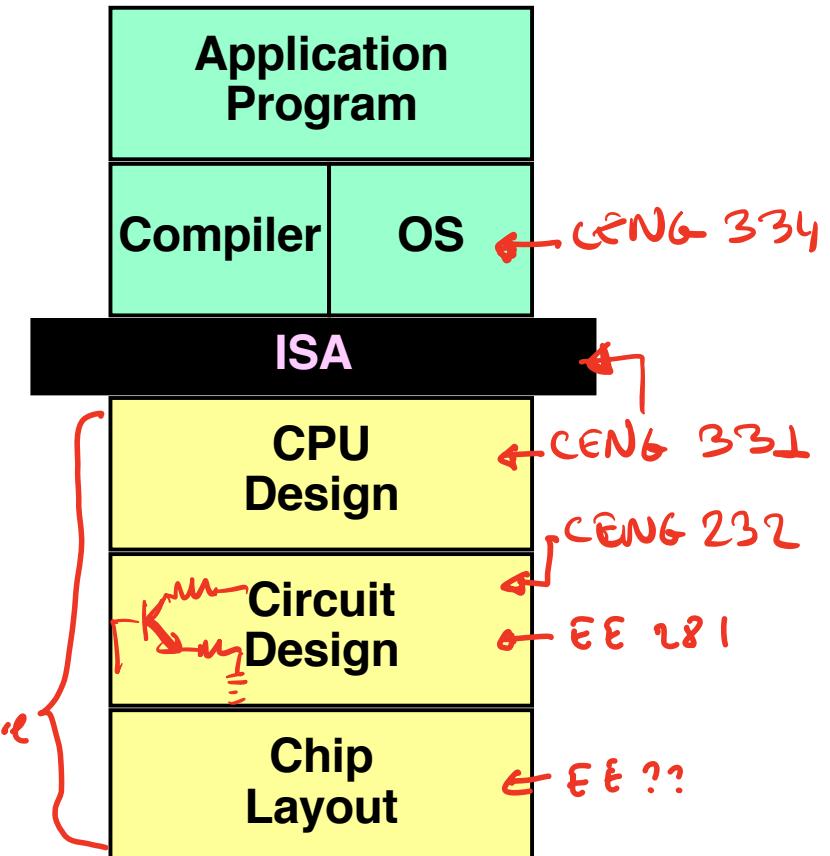
# Instruction Set Architecture

## ■ Assembly Language View

- Processor state
  - Registers, memory, ...
- Instructions
  - **addq, pushq, ret, ...**
  - How instructions are encoded as bytes

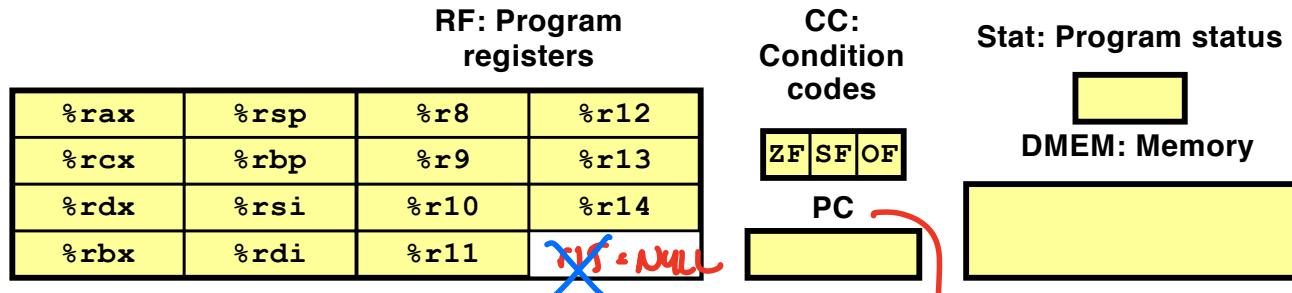
## ■ Layer of Abstraction

- Above: how to program machine
  - Processor executes instructions in a sequence
- Below: what needs to be built
  - Use variety of tricks to make it run fast
  - E.g., execute multiple instructions simultaneously



~~LSA~~

# Y86-64 Processor State



- Program Registers
  - 15 registers (omit %r15). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero
    - SF:Negative
    - OF: Overflow
- Program Counter *rip*
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

# Y86-64 Instruction Set #1

Binary Encoding of Instruction

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Y86-64 Instructions

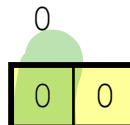
## ■ Format

- 1–10 bytes of information read from memory
  - Can determine instruction length from first byte
  - Not as many instruction types, and simpler encoding than with x86-64
- Each accesses and modifies some part(s) of the program state

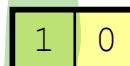
# Y86-64 Instruction Set #2

**Byte**

halt



nop



cmoveXX rA, rB



irmovq V, rB



rmmovq rA, D(rB)



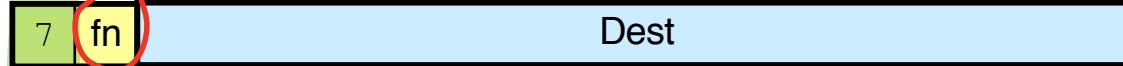
mrmovq D(rB), rA



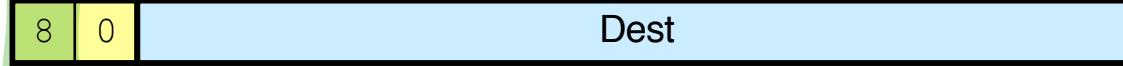
OPq rA, rB



jXX Dest



call Dest



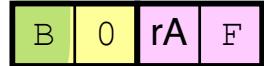
ret



pushq rA



popq rA

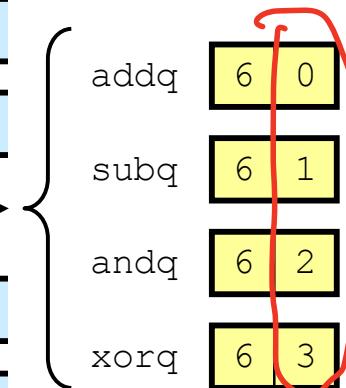


*cmoveq : unconditional  
conditional  
mov*

rrmovq	2 0
cmovele	2 1
cmovl	2 2
cmove	2 3
cmovne	2 4
cmovge	2 5
cmovg	2 6

# Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						



# Y86-64 Instruction Set #4

Byte	0	1	2	3	4	5	6	7	
halt	0	0							jmp
nop	1	0							jle
cmoveXX rA, rB	2	fn	rA	rB					jl
irmovq V, rB	3	0	F	rB	V				je
rmmovq rA, D(rB)	4	0	rA	rB	D				jne
mrmovq D(rB), rA	5	0	rA	rB	D				jge
OPq rA, rB	6	fn	rA	rB					jg
jXX Dest	7	fn			Dest				
call Dest	8	0			Dest				
ret	9	0							
pushq rA	A	0	rA	F					NO REGISTER
popq rA	B	0	rA	F					

# Encoding Registers

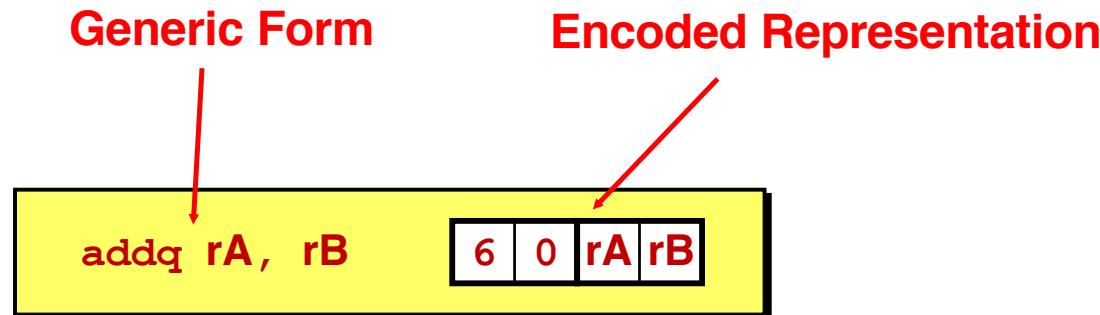
- Each register has 4-bit ID

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7
%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

- Same encoding as in x86-64
- Register ID 15 (0xF) indicates “no register”
  - Will use this in our hardware design in multiple places

# Instruction Example

## ■ Addition Instruction



%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7
No Register	F

- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- e.g.,  $\text{addq } \%rax, \%rsi$  Encoding:
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

addq	6	0
subq	6	1
andq	6	2
xorq	6	3

# Arithmetic and Logical Operations

## Instruction Code

Add



## Function Code

Subtract (rA from rB)



And



Exclusive-Or



- Refer to generically as “OPq”
- Encodings differ only by “function code”
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

# Move Operations



"unconditional conditional move"  $\equiv$  Register  $\rightarrow$  Register

`rrmovq rA, rB`

2	0	rA	rB
---	---	----	----

Register  $\rightarrow$  Register

`irmovq V, rB`

3	0	F	rB
---	---	---	----

Immediate  $\rightarrow$  Register

V

`rmmovq rA, D(rB)`

4	0	rA	rB
---	---	----	----

Displacement

Register  $\rightarrow$  Memory

`mrmovq D(rB), rA`

5	0	rA	rB
---	---	----	----

Memory  $\rightarrow$  Register

D

- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

# Move Instruction Examples

X86-64

```
movq $0xabcd, %rdx
```

Encoding:

Y86-64

```
irmovq $0xabcd, %rdx
```

30 F2 cd ab 00 00 00 00 00 → Little endian  
8-byte

```
movq %rsp, %rbx
```

Encoding:

```
rrmovq %rsp, %rbx
```

20 43 C

```
movq -12(%rbp), %rcx
```

Encoding:

```
mrmovq -12(%rbp), %rcx
```

50 15 f4 ff ff ff ff ff ff ff

-12

```
movq %rsi, 0x41c(%rsp)
```

Encoding:

```
rmmovq %rsi, 0x41c(%rsp)
```

40 64 1c 04 00 00 00 00 00

# Conditional Move Instructions

Move Unconditionally

<i>cmoveq</i>	<code>rrmovq rA, rB</code>	
<b>Move When Less or Equal</b>		
	<code>cmovle rA, rB</code>	
<b>Move When Less</b>		
	<code>cmovl rA, rB</code>	
<b>Move When Equal</b>		
	<code>cmove rA, rB</code>	
<b>Move When Not Equal</b>		
	<code>cmovne rA, rB</code>	
<b>Move When Greater or Equal</b>		
	<code>cmovge rA, rB</code>	
<b>Move When Greater</b>		
	<code>cmovg rA, rB</code>	

- Refer to generically as “cmovXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - (Conditionally) copy value from source to destination register

# Jump Instructions

Jump (Conditionally)



*target address to jump if condition is satisfied*

- Refer to generically as “jXX”
- Encodings differ only by “function code” fn
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode absolute full destination address
  - Unlike PC-relative addressing seen in x86-64
    - Note: PC stands for Program Counter
    - In x86 we have %rip : Instruction Pointer

*target = %rip + ↗*

# Jump Instructions

## Jump Unconditionally

**jmp Dest**

7	0
---	---

 Dest

## Jump When Less or Equal

**jle Dest**

7	1
---	---

 Dest

## Jump When Less

**jl Dest**

7	2
---	---

 Dest

## Jump When Equal

**je Dest**

7	3
---	---

 Dest

## Jump When Not Equal

**jne Dest**

7	4
---	---

 Dest

## Jump When Greater or Equal

**jge Dest**

7	5
---	---

 Dest

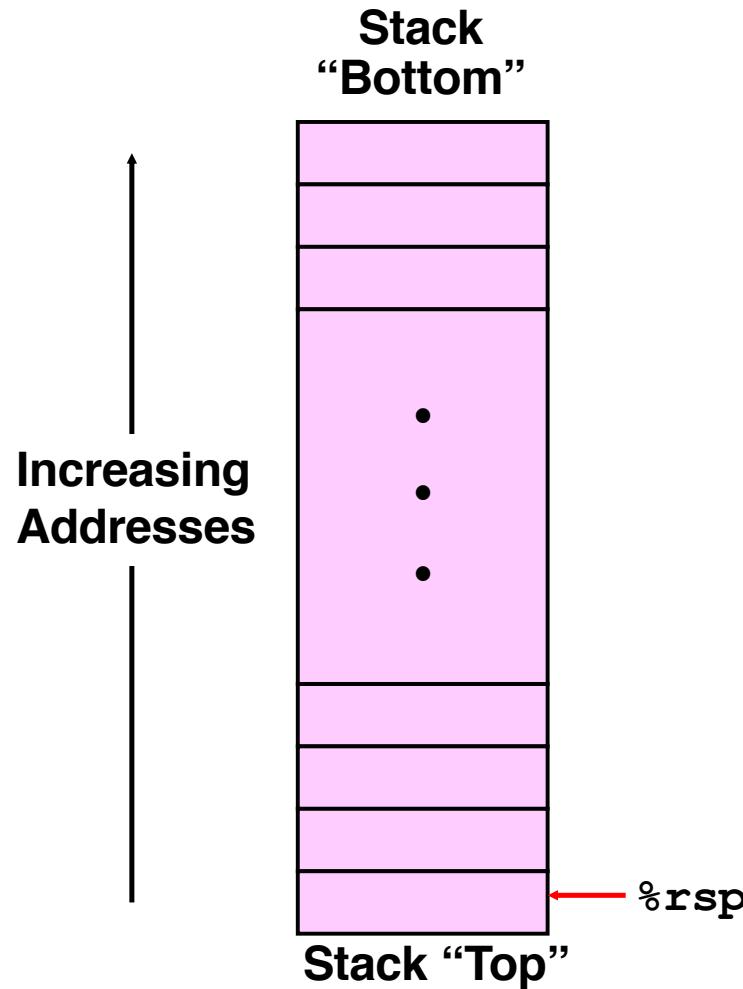
## Jump When Greater

**jg Dest**

7	6
---	---

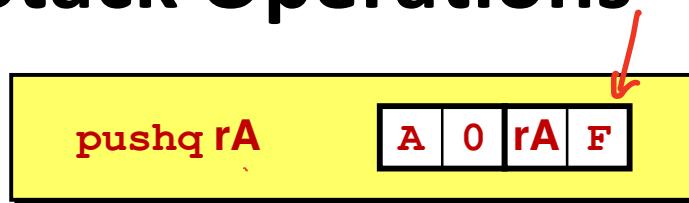
 Dest

# Y86-64 Program Stack

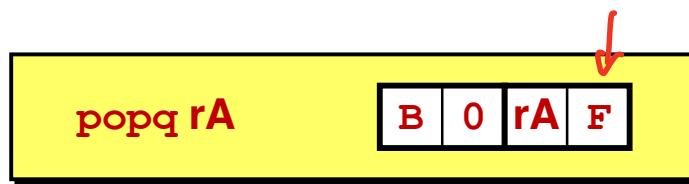


- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by `%rsp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer

# Stack Operations



- Decrement  $\%rsp$  by 8
- Store word from  $rA$  to memory at  $\%rsp$
- Like x86-64



- Read word from memory at  $\%rsp$
- Save in  $rA$
- Increment  $\%rsp$  by 8
- Like x86-64

# Subroutine Call and Return

call Dest

8	0
---	---

Dest

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

return address

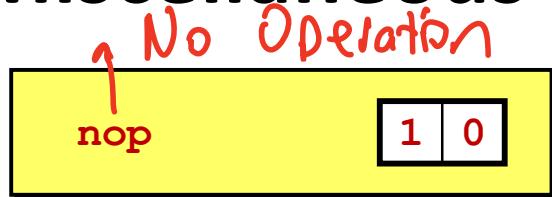
ret

9	0
---	---

- Pop value from stack
- Use as address for next instruction
- Like x86-64

return address

# Miscellaneous Instructions



- Don't do anything



- Stop executing instructions
- x86-64 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

# Status Conditions

All  
OK

Mnemonic	Code
AOK	1

Mnemonic	Code
HLT	2

Mnemonic	Code
ADR	3

Mnemonic	Code
INS	4

- Normal operation
- Halt instruction encountered
- Bad address (either instruction or data) encountered
- Invalid instruction encountered
- Desired Behavior
  - If AOK, keep going
  - Otherwise, stop program execution

# Writing Y86-64 Code

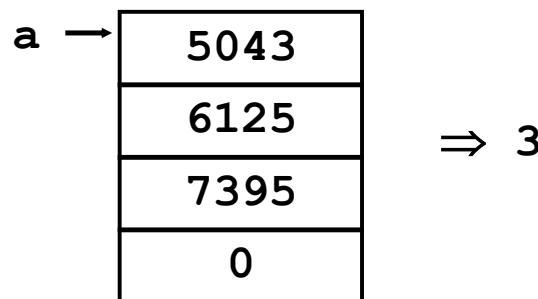
## ■ Try to Use C Compiler as Much as Possible

- Write code in C
- Compile for x86-64 with `gcc -Og -S`
- Transliterate into Y86-64
- *Modern compilers make this more difficult*

## ■ Coding Example

- Find number of elements in null-terminated list

```
int len1(int a[]);
```



# Y86-64 Code Generation Example

## ■ First Try

- Write typical array code

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
    long len;
    for (len = 0; a[len]; len++)
        ;
    return len;
}
```

## ■ Problem

- Hard to do array indexing on Y86-64
  - Since don't have scaled addressing modes

L3:

```
addq $1,%rax
cmpq $0, (%rdi,%rax,8)
jne L3
```

- Compile with gcc -Og -S

# Y86-64 Code Generation Example #2

## ■ Second Try

- Write C code that mimics expected Y86-64 code

```
long len2(long *a)
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        len++;
        val = *(long *) ip;
    }
    return len;
}
```

## ■ Result

- Compiler generates exact same code as before!
- Compiler converts both versions into same intermediate form

# Y86-64 Code Generation Example #3

```

len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9           # Constant 8
    irmovq $0, %rax          # len = 0
    mrmovq (%rdi), %rdx      # val = *a
    andq %rdx, %rdx          # Test val
    je Done                  # If zero, goto Done

Loop:
    addq %r8, %rax          # len++
    addq %r9, %rdi           # a++
    mrmovq (%rdi), %rdx      # val = *a
    andq %rdx, %rdx          # Test val
    jne Loop                 # If !0, goto Loop

Done:
    ret

```

Register	Use
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

# Y86-64 Sample Program Structure #1

```
init:          # Initialization
  ...
  call Main
  halt

  .align 8      # Program data
array:

Main:         # Main function
  ...
  call len    ...

len:          # Length function
  ...

  .pos 0x100   # Placement of stack
Stack:
```

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data

# Y86-64 Program Structure #2

```
init:  
    # Set up stack pointer  
    irmovq Stack, %rsp  
    # Execute main program  
    call Main  
    # Terminate  
    halt  
  
# Array of 4 elements + terminating 0  
.align 8  
Array:  
.quad 0x000d000d000d000d  
.quad 0x00c000c000c000c0  
.quad 0x0b000b000b000b00  
.quad 0xa000a000a000a000  
.quad 0
```

- Program starts at address 0
- Must set up stack
- Must initialize data
- Can use symbolic names

# Y86-64 Program Structure #3

Main:

```
    irmovq array,%rdi
    # call len(array)
    call len
    ret
```

## ■ Set up call to len

- Follow x86-64 procedure conventions
- Push array address as argument

# Assembling Y86-64 Program $\xrightarrow{yas!}$ object file

unix> yas len.ys

*text code in Y86*

- Generates “object code” file len.yo
  - Actually looks like disassembler output

```

0x054:                                | len:
0x054: 30f801000000000000000000 |    irmovq $1, %r8          # Constant 1
0x05e: 30f908000000000000000000 |    irmovq $8, %r9          # Constant 8
0x068: 30f000000000000000000000 |    irmovq $0, %rax         # len = 0
0x072: 502700000000000000000000 |    mrmovq (%rdi), %rdx     # val = *a
0x07c: 6222                            |    andq %rdx, %rdx        # Test val
0x07e: 73a000000000000000000000 |    je Done                 # If zero, goto Done
0x087:                                | Loop:
0x087: 6080                            |    addq %r8, %rax        # len++
0x089: 6097                            |    addq %r9, %rdi        # a++
0x08b: 502700000000000000000000 |    mrmovq (%rdi), %rdx     # val = *a
0x095: 6222                            |    andq %rdx, %rdx        # Test val
0x097: 748700000000000000000000 |    jne Loop                # If != 0, goto Loop
0x0a0:                                | Done:
0x0a0: 90                             |    ret

```

# Simulating Y86-64 Program

```
unix> yis len.yo
```

- Instruction set simulator

- Computes effect of each instruction on processor state
- Prints changes in state from original

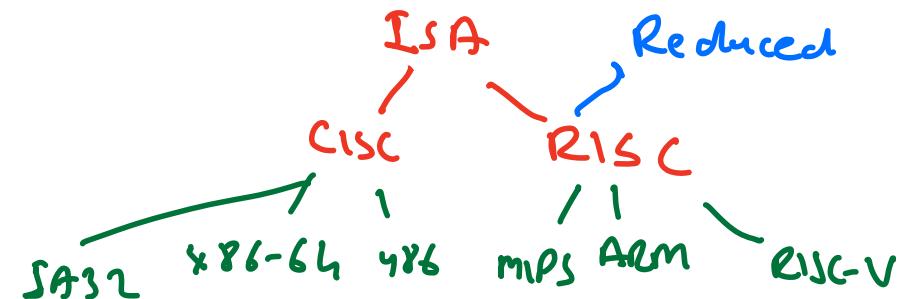
```
Stopped in 33 steps at PC = 0x13, Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax: 0x0000000000000000          0x0000000000000004
%rsp: 0x0000000000000000          0x0000000000000100
%rdi: 0x0000000000000000          0x0000000000000038
%r8: 0x0000000000000000          0x0000000000000001
%r9: 0x0000000000000000          0x0000000000000008

Changes to memory:
0x00f0: 0x0000000000000000      0x0000000000000053
0x00f8: 0x0000000000000000      0x0000000000000001
```

Complex Instruction Set Computer

# CISC Instruction Sets

- Complex Instruction Set Computer
- IA32 is example  $\times 86-64$
- $\times 86-64$  32-bit predecessor of



## ■ Stack-oriented instruction set

- Use stack to pass arguments, save program counter
- Explicit push and pop instructions

## ■ Arithmetic instructions can access memory

- `addq %rax, 12(%rbx,%rcx,8)`
  - requires memory read and write  $\rightarrow M[\%rbx + 8 \cdot \%rcx + 12] += \%rax$
  - Complex address calculation

## ■ Condition codes

- Set as side effect of arithmetic and logical instructions

## ■ Philosophy

- Add instructions to perform “typical” programming tasks

# Reduced RISC Instruction Sets

RISC  $\longleftrightarrow$  CISC

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
- **Fewer, simpler instructions**
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- **Register-oriented instruction set**
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- **Only load and store instructions can access memory**
  - Similar to Y86-64 `mrmovq` and `rmmovq`
- **No Condition codes**
  - Test instructions return 0/1 in register

→ RISC Roles

# MIPS Registers

\$0	\$0	Constant 0
\$1	\$at	Reserved Temp.
\$2	\$v0	Return Values
\$3	\$v1	
\$4	\$a0	Procedure arguments
\$5	\$a1	
\$6	\$a2	
\$7	\$a3	
\$8	\$t0	
\$9	\$t1	
\$10	\$t2	Caller Save Temporaries: May be overwritten by called procedures
\$11	\$t3	
\$12	\$t4	
\$13	\$t5	
\$14	\$t6	
\$15	\$t7	
\$16	\$s0	Callee Save Temporaries: May not be overwritten by called procedures
\$17	\$s1	
\$18	\$s2	
\$19	\$s3	
\$20	\$s4	
\$21	\$s5	
\$22	\$s6	
\$23	\$s7	Caller Save Temp
\$24	\$t8	
\$25	\$t9	
\$26	\$k0	Reserved for Operating Sys
\$27	\$k1	
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$s8	Callee Save Temp
\$31	\$ra	Return Address

# MIPS Instruction Examples

All instructions have the same length

## R-R

Op	Ra	Rb	Rd	00000	Fn
----	----	----	----	-------	----

addu \$3,\$2,\$1 *Registers* # Register add: \$3 = \$2+\$1

## R-I

Op	Ra	Rb	Immediate
----	----	----	-----------

addu \$3,\$2, 3145 # Immediate add: \$3 = \$2+3145  
sll \$3,\$2,2 # Shift left: \$3 = \$2 << 2

## Branch

Op	Ra	Rb	Offset
----	----	----	--------

beq \$3,\$2,dest # Branch when \$3 = \$2

## Load/Store

Op	Ra	Rb	Offset
----	----	----	--------

lw \$3,16(\$2) # Load Word: \$3 = M[\$2+16]  
sw \$3,16(\$2) # Store Word: M[\$2+16] = \$3

No other instructions can access memory

# CISC vs. RISC

ARM → ISA

## ■ Original Debate

- Strong opinions!
- CISC proponents---easy for compiler, fewer code bytes
- RISC proponents---better for optimizing compilers, can make run fast with simple chip design

## ■ Current Status

- For desktop processors, choice of ISA not a technical issue
  - With enough hardware, can make anything run fast
  - Code compatibility more important
- x86-64 adopted many RISC features
  - More registers; use them for argument passing
- For embedded processors, RISC makes sense
  - Smaller, cheaper, less power
  - Most cell phones use ARM processor

x86-64  
↑  
IA32 - CISC

# Summary

## ■ Y86-64 Instruction Set Architecture

- Similar state and instructions as x86-64
- Simpler encodings
- Somewhere between CISC and RISC

## ■ How Important is ISA Design?

- Less now than before
  - With enough hardware, can make almost anything go fast

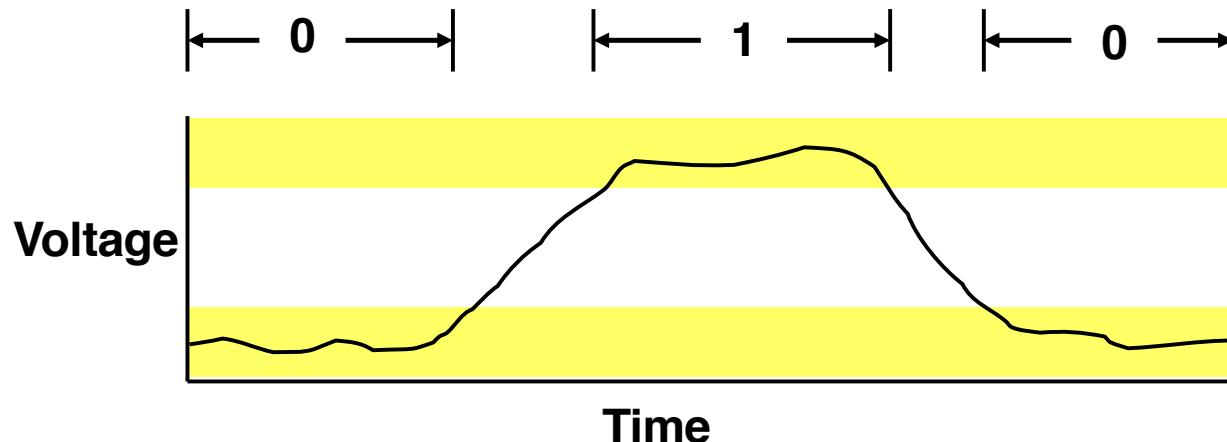
↳ ARM

# Processor Architecture: Logic Design

# Overview of Logic Design

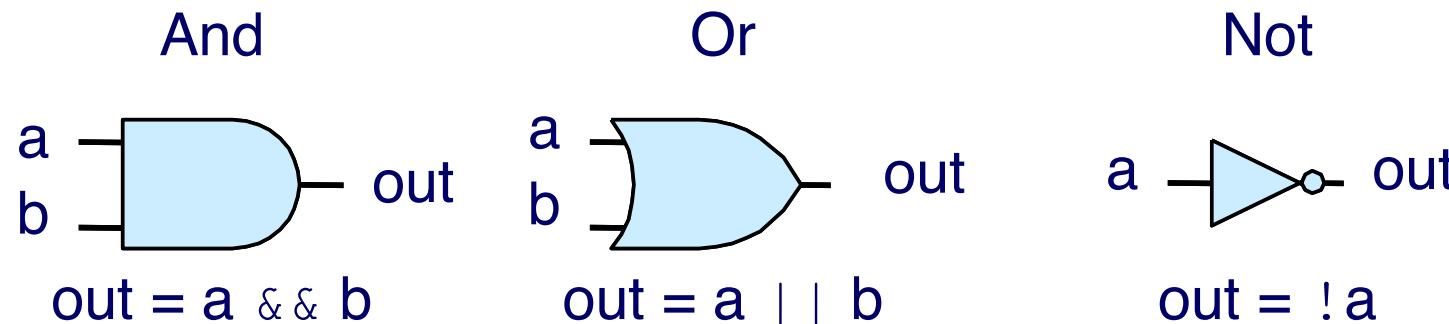
- **Fundamental Hardware Requirements**
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage
- **Bits are Our Friends**
  - Everything expressed in terms of values 0 and 1
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
    - Store bits of information

# Digital Signals

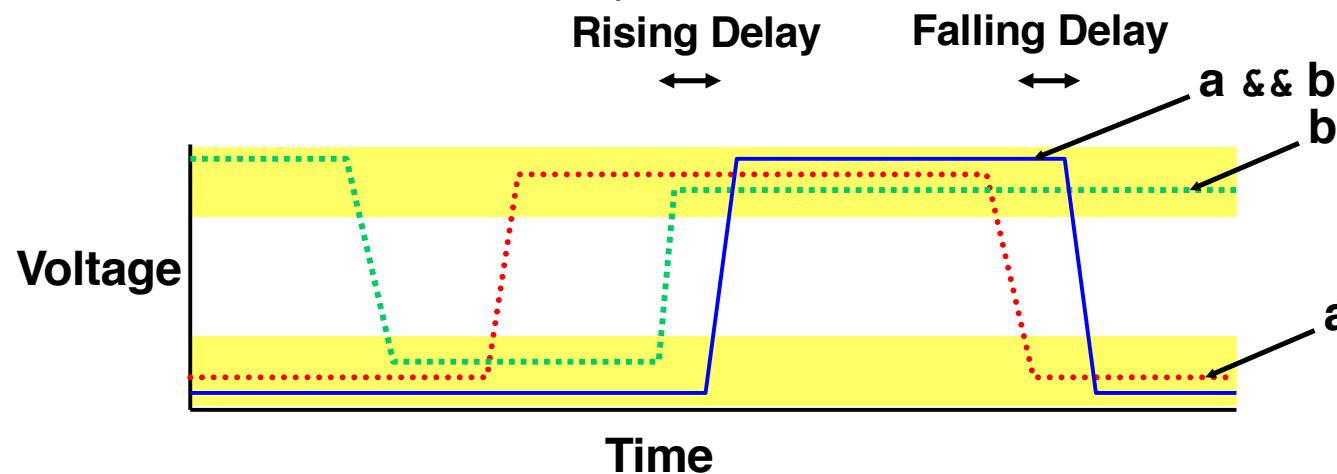


- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

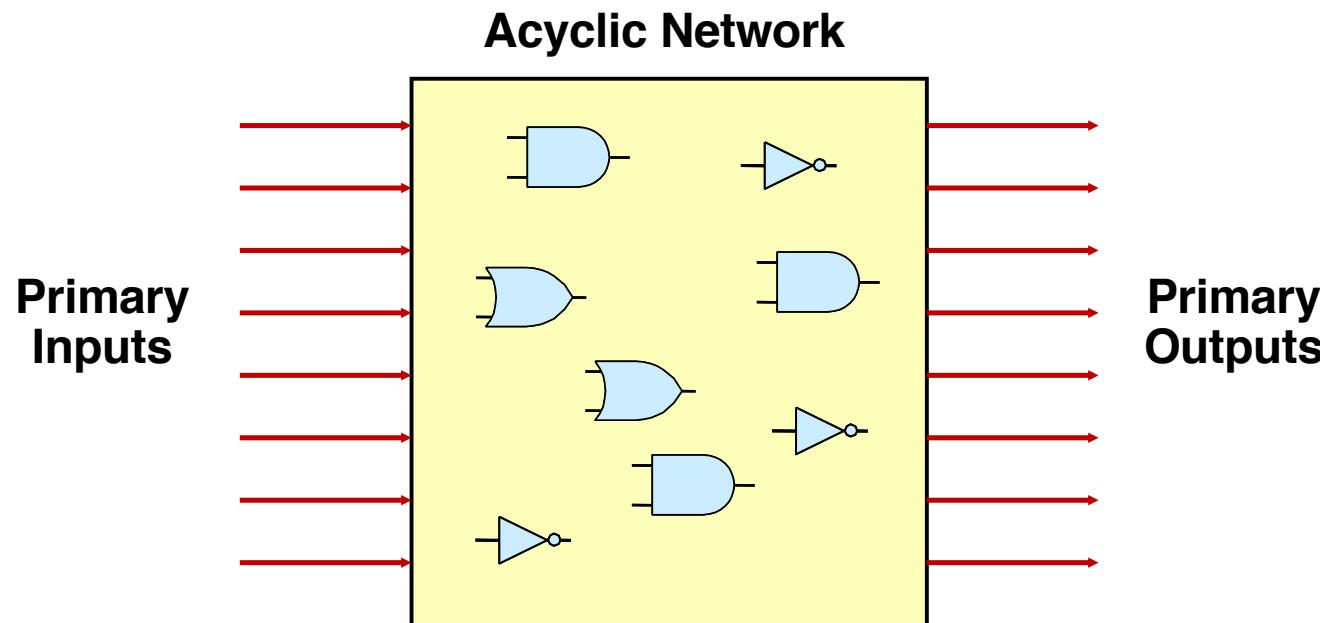
# Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some, small delay



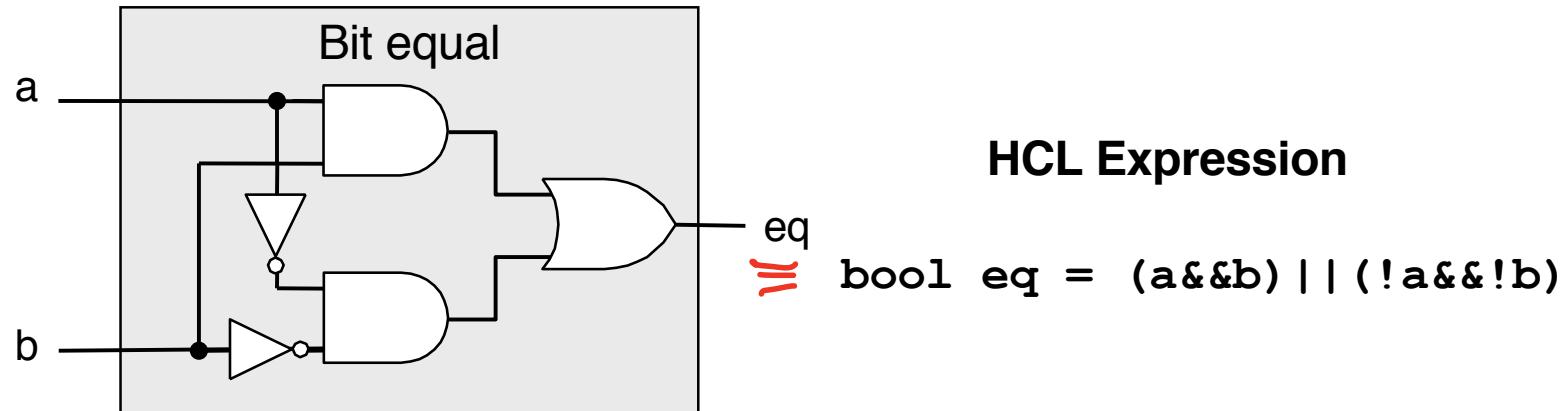
# Combinational Circuits



## ■ Acyclic Network of Logic Gates

- Continuously responds to changes on primary inputs
- Primary outputs become (after some delay) Boolean functions of primary inputs

# Bit Equality

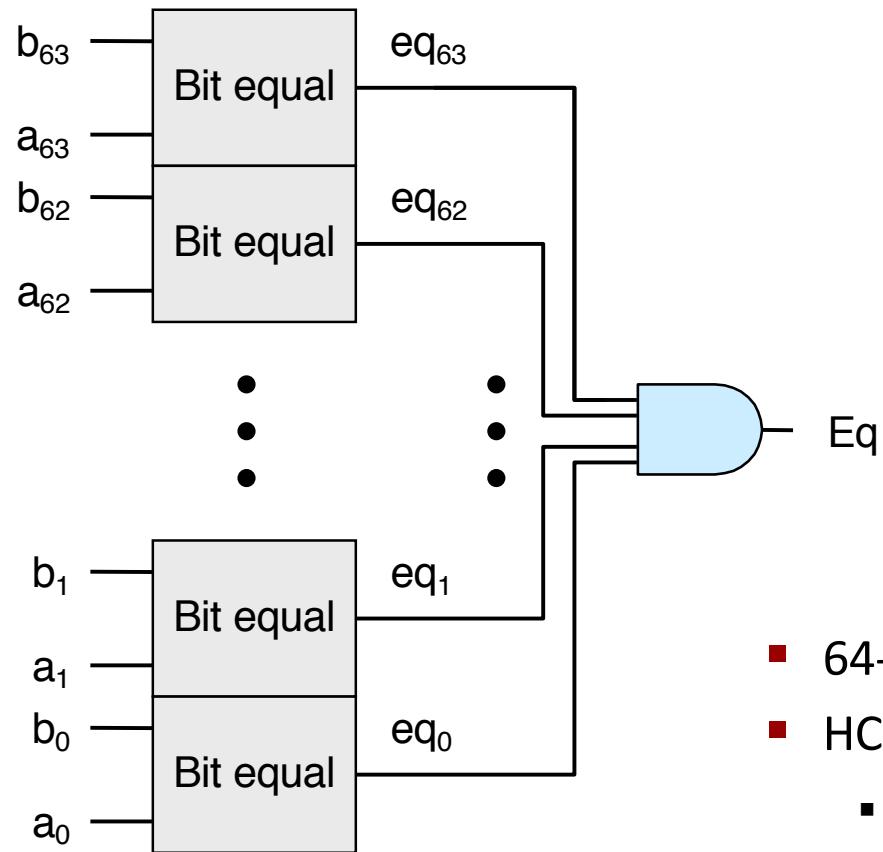


- Generate 1 if *a* and *b* are equal

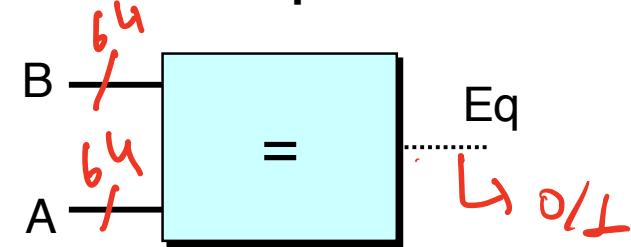
## ■ Hardware Control Language (HCL)

- Very simple hardware description language
  - Boolean operations have syntax similar to C logical operations
- We'll use it to describe control logic for processors

# Word Equality



## Word-Level Representation



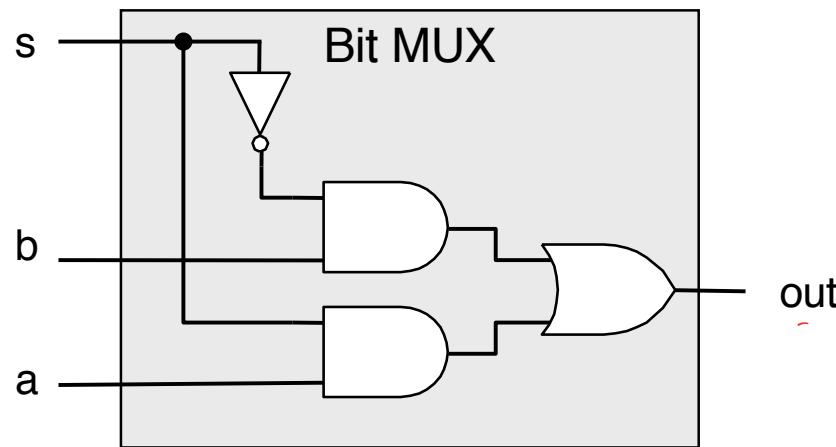
## HCL Representation

```
bool Eq = (A == B)
```

- 64-bit word size
- HCL representation
  - Equality operation
  - Generates Boolean value

# Bit-Level Multiplexor

Select signal



## HCL Expression

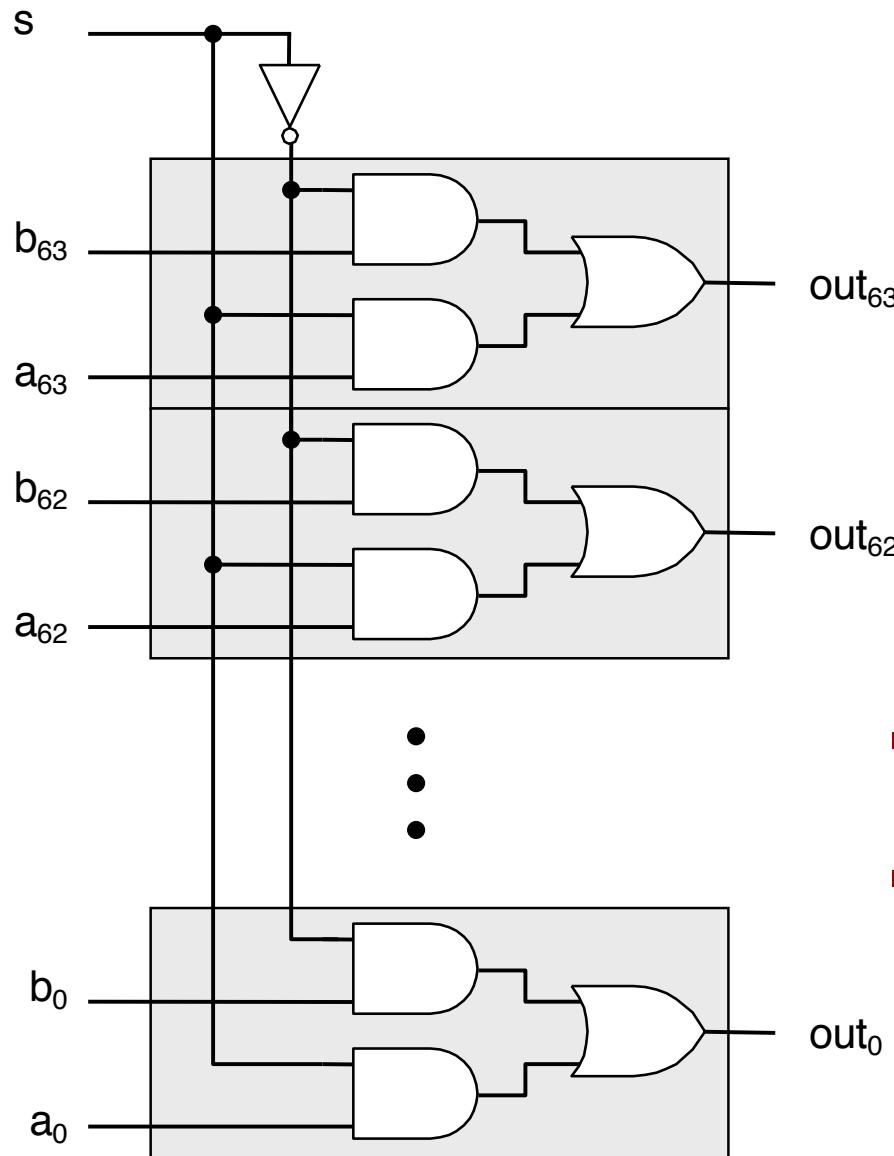
```
bool out = (s&&a) || (!s&&b)
```

$$s=1 \Rightarrow out = a$$

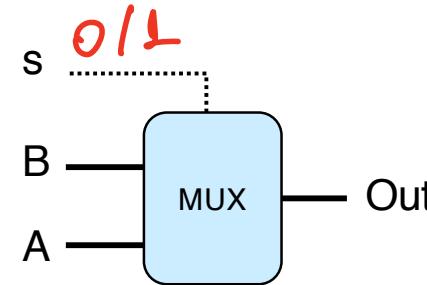
$$s=0 \Rightarrow out = b$$

- Control signal s
- Data signals a and b
- Output a when  $s=1$ , b when  $s=0$

# Word Multiplexor



## Word-Level Representation



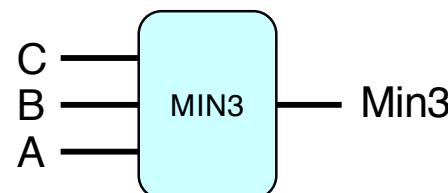
## HCL Representation

```
int Out = [
    s : A;
    1 : B;
];
```

- Select input word A or B depending on control signal  $s$
- HCL representation
  - Case expression
  - Series of test : value pairs
  - Output value for first successful test

# HCL Word-Level Examples

## Minimum of 3 Words

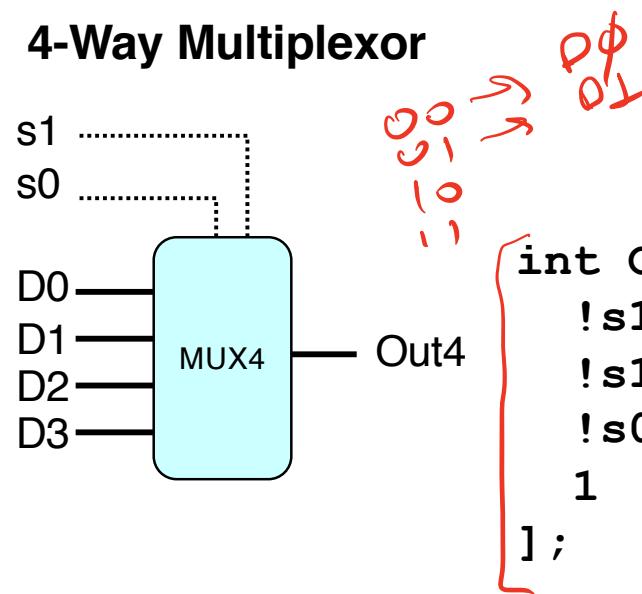


```

int Min3 = [
    A < B && A < C : A;
    B < A && B < C : B;
    1
    : C;
];
  
```

- Find minimum of three input words
- HCL case expression
- Final case guarantees match

## 4-Way Multiplexor

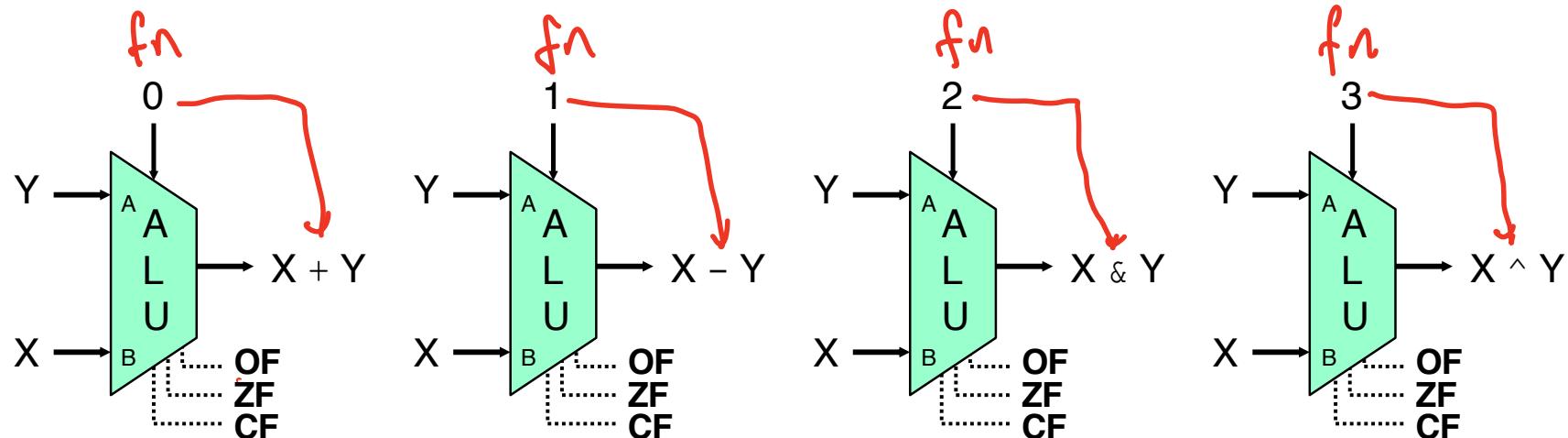


```

int Out4 = [
    !s1&&!s0: D0;
    !s1&&s1: D1;
    !s0 : D2;
    1 : D3;
];
  
```

- Select one of 4 inputs based on two control bits
- HCL case expression
- Simplify tests by assuming sequential matching

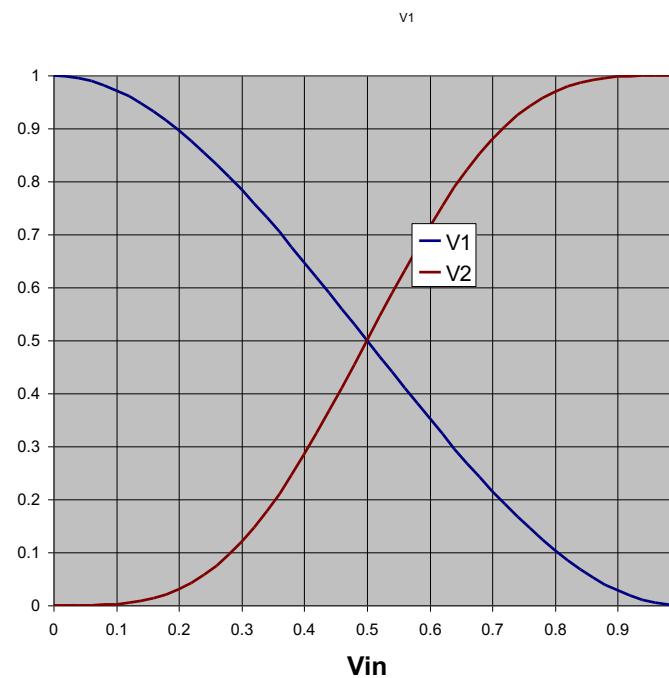
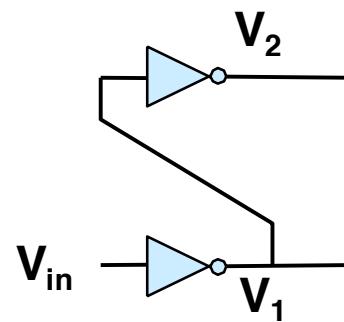
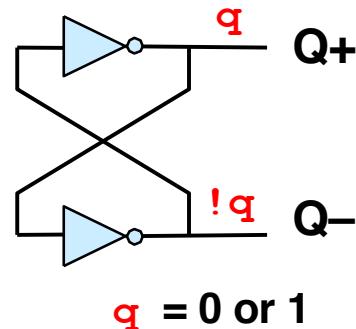
# Arithmetic Logic Unit



- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

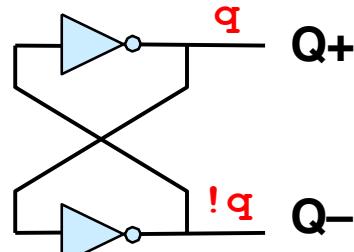
# Storing 1 Bit

## Bistable Element



# Storing 1 Bit (cont.)

## Bistable Element



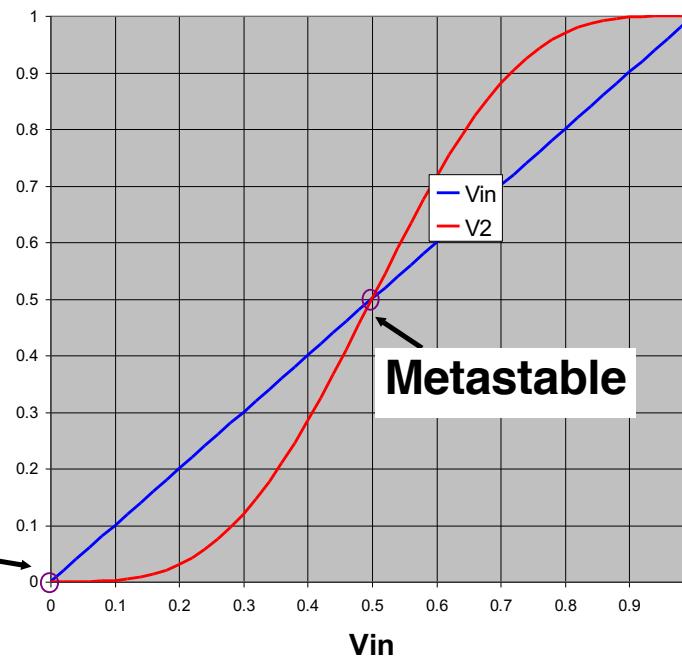
$$q = 0 \text{ or } 1$$

## Stable 1

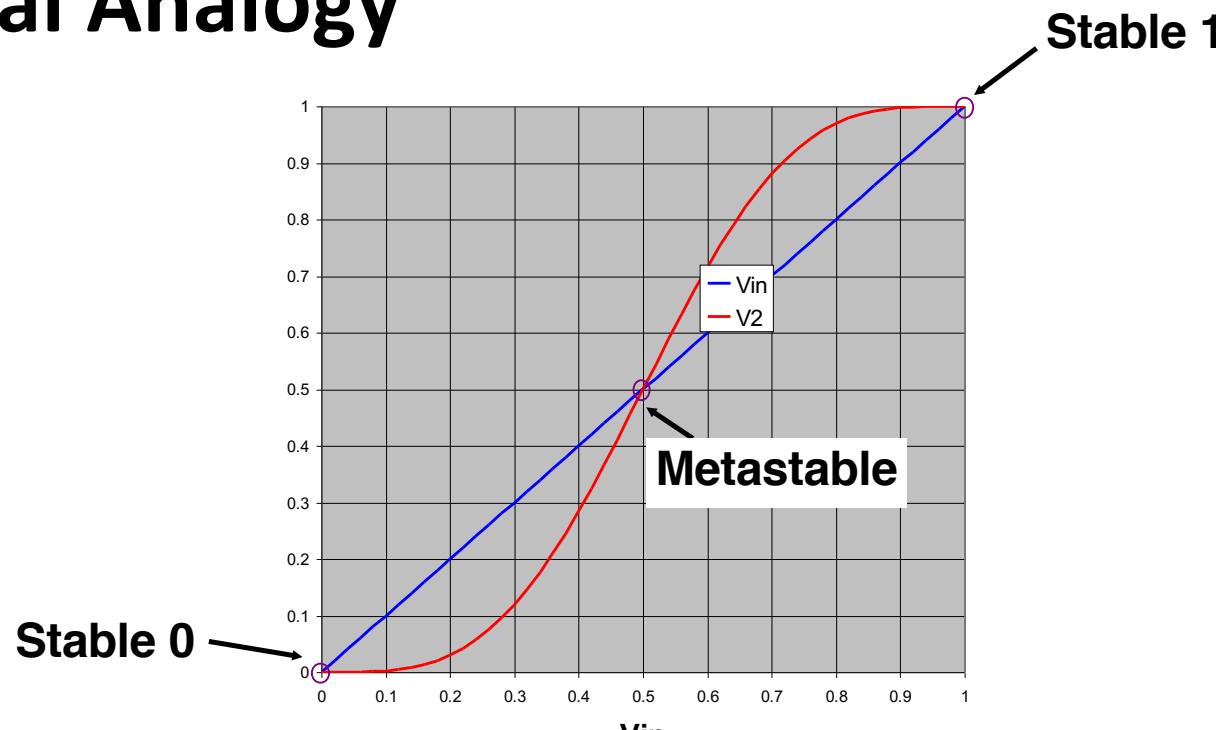
$$V_{in} = V_2$$

$V_2$

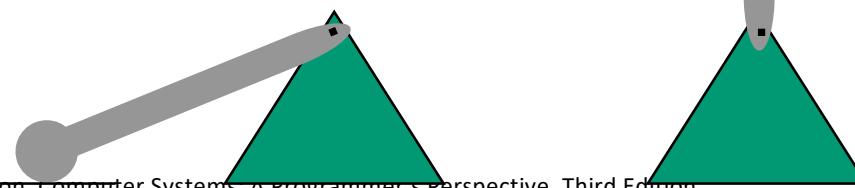
## Stable 0



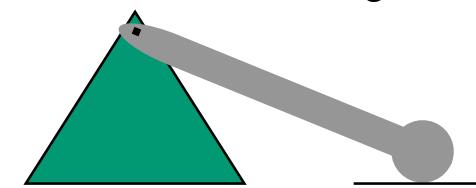
# Physical Analogy



Stable left

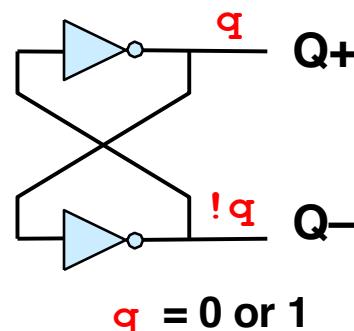


Stable right

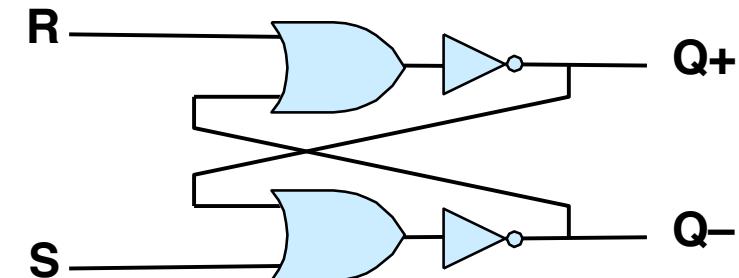


# Storing and Accessing 1 Bit

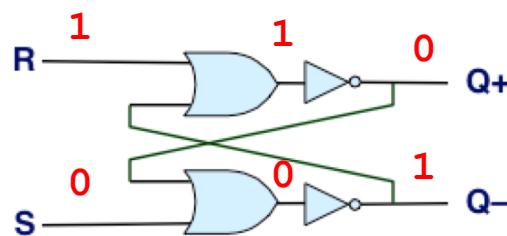
## Bistable Element



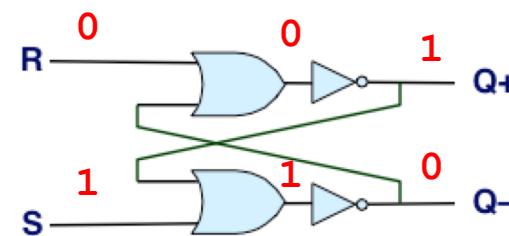
## R-S Latch



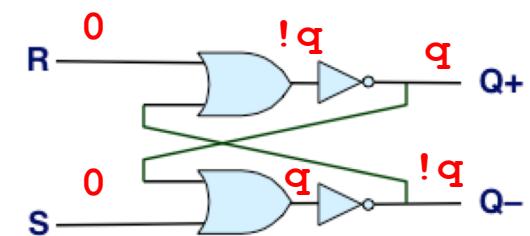
## Resetting



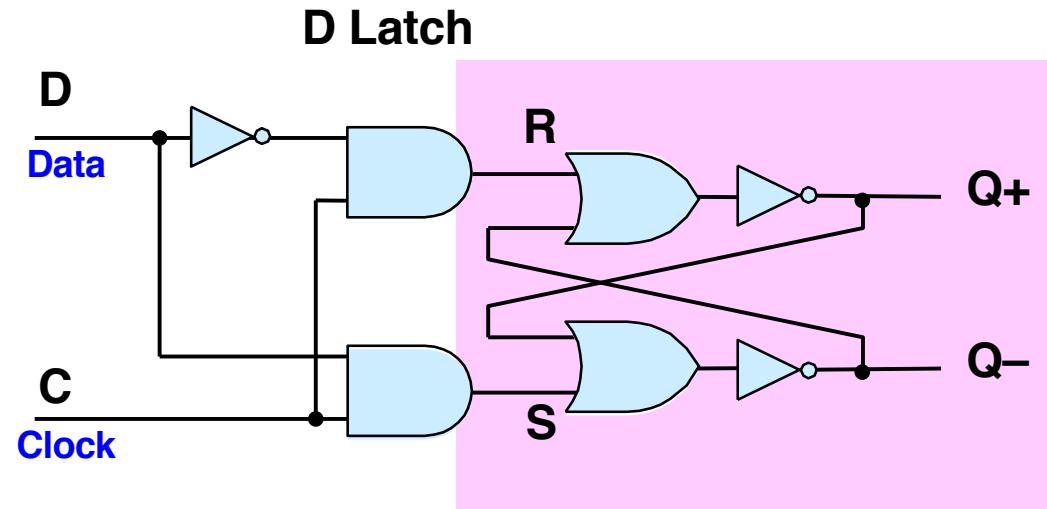
## Setting



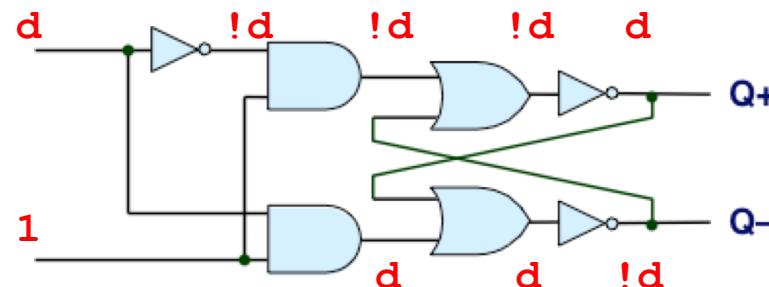
## Storing



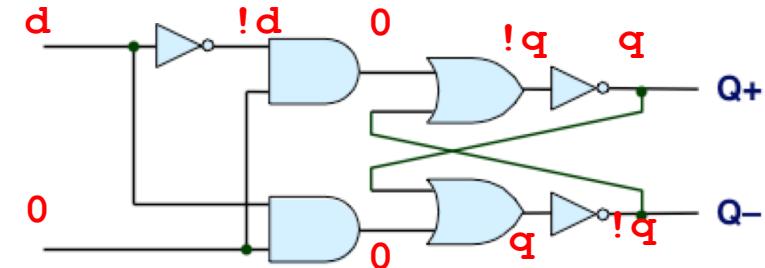
# 1-Bit Latch



**Latching**

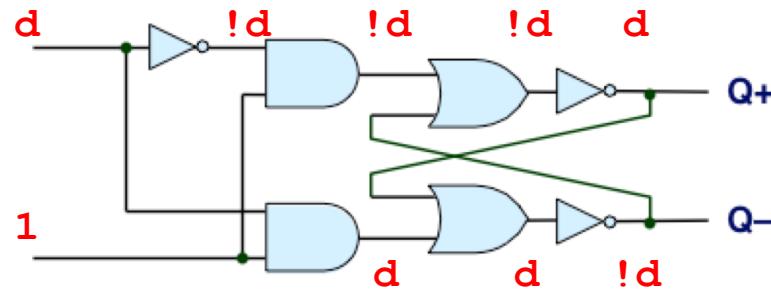


**Storing**

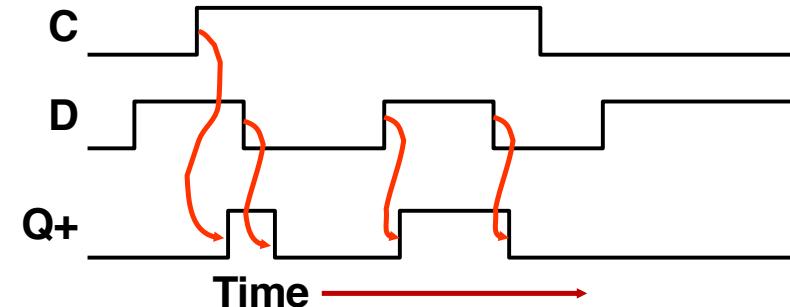


# Transparent 1-Bit Latch

## Latching

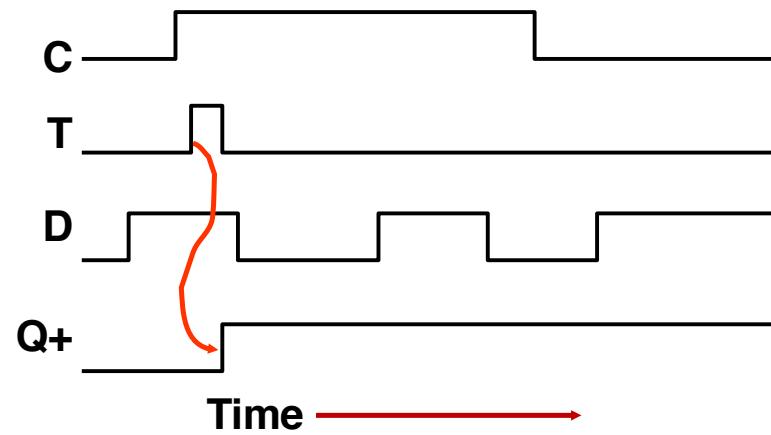
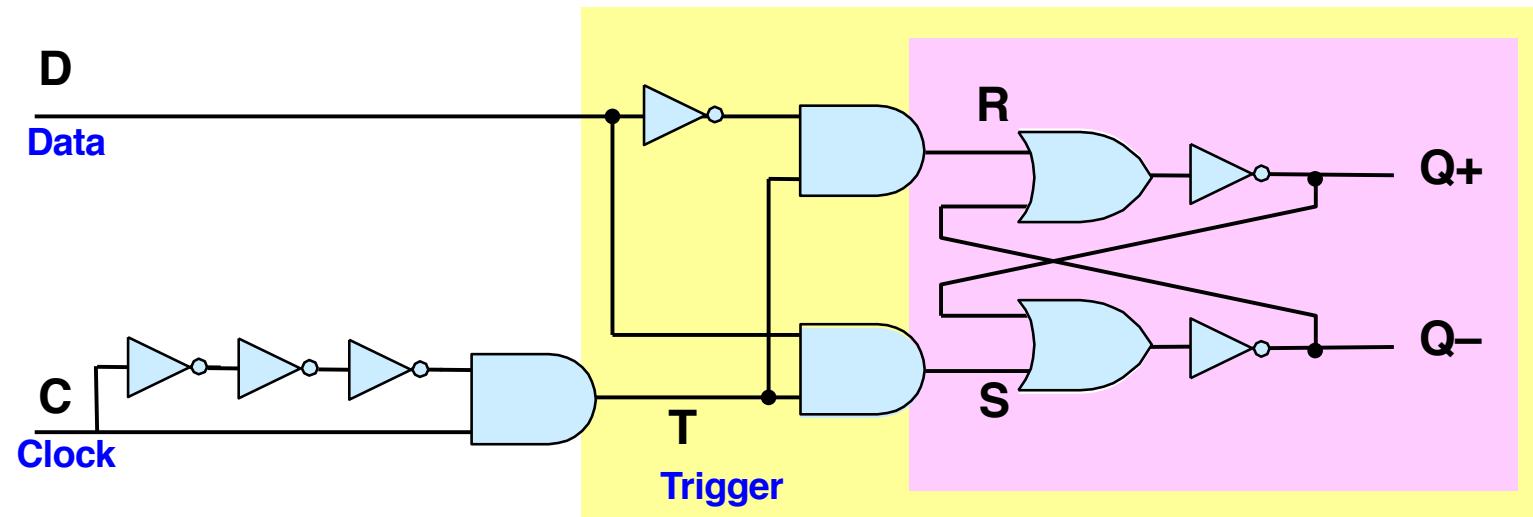


## Changing D



- When in latching mode, combinational propagation from D to Q+ and Q-
- Value latched depends on value of D as C falls

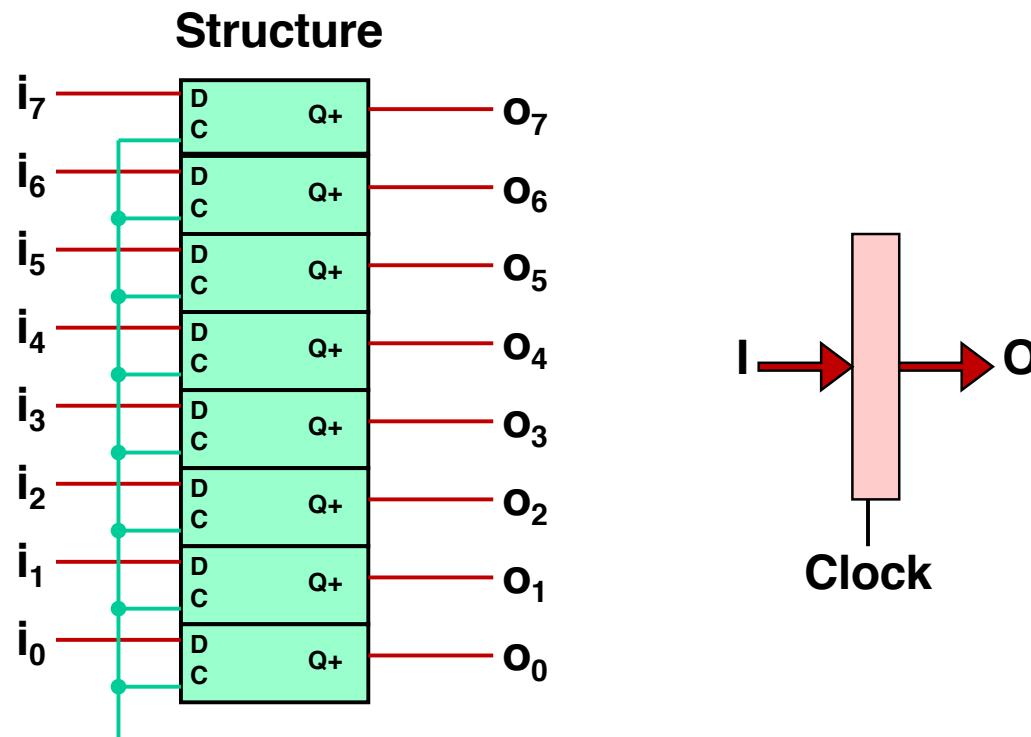
# Edge-Triggered Latch



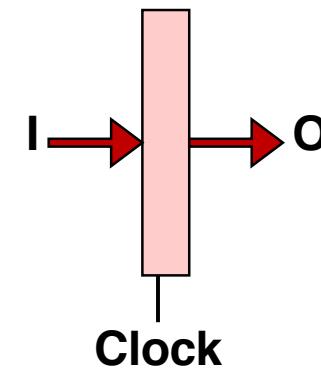
- Only in latching mode for brief period
  - Rising clock edge
- Value latched depends on data as clock rises
- Output remains stable at all other times

# Registers

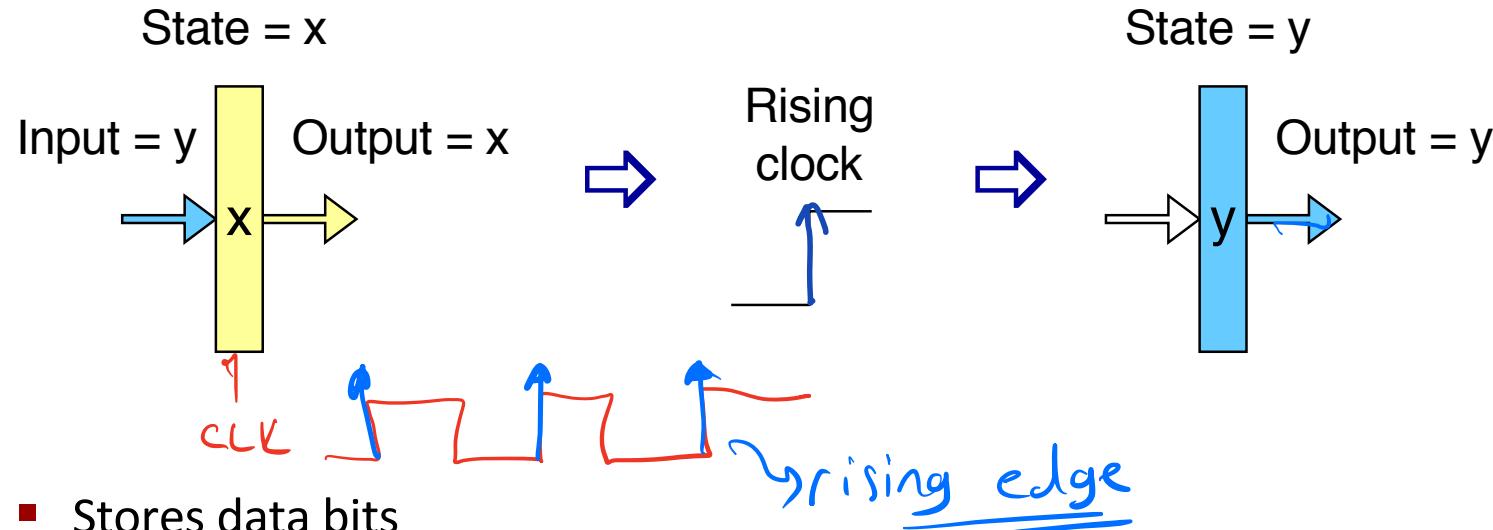
*rising edge*



- Stores word of data
  - Different from *program registers* seen in assembly code
- Collection of *edge-triggered latches*
- Loads input on *rising edge* of clock

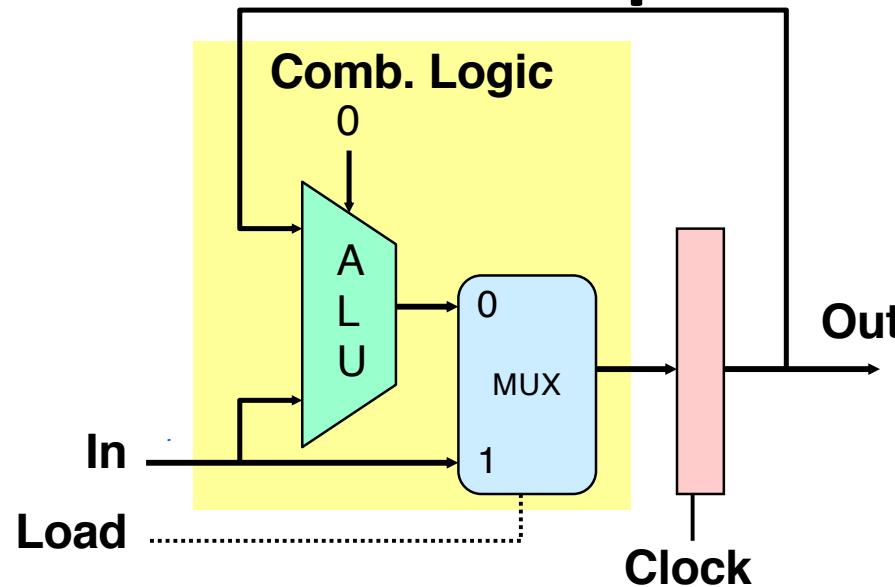


# Register Operation

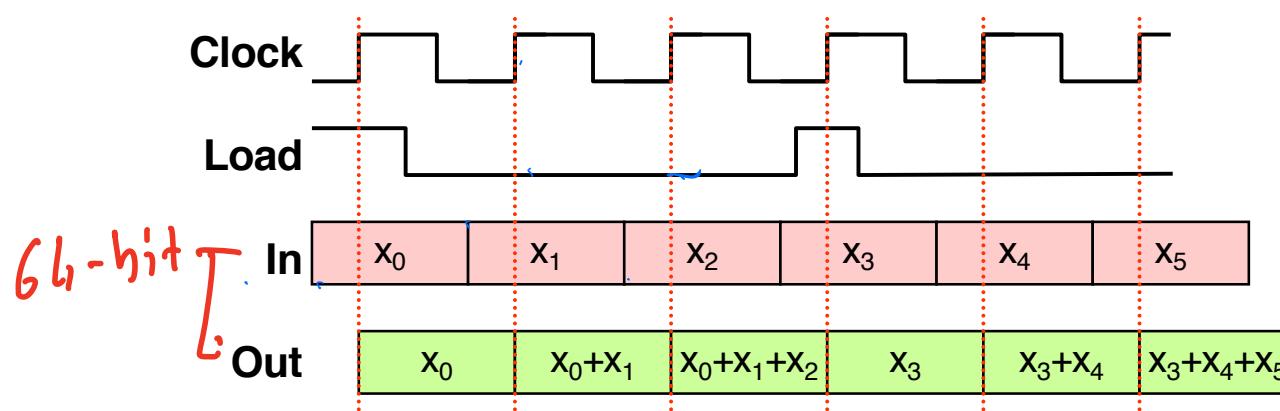


- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

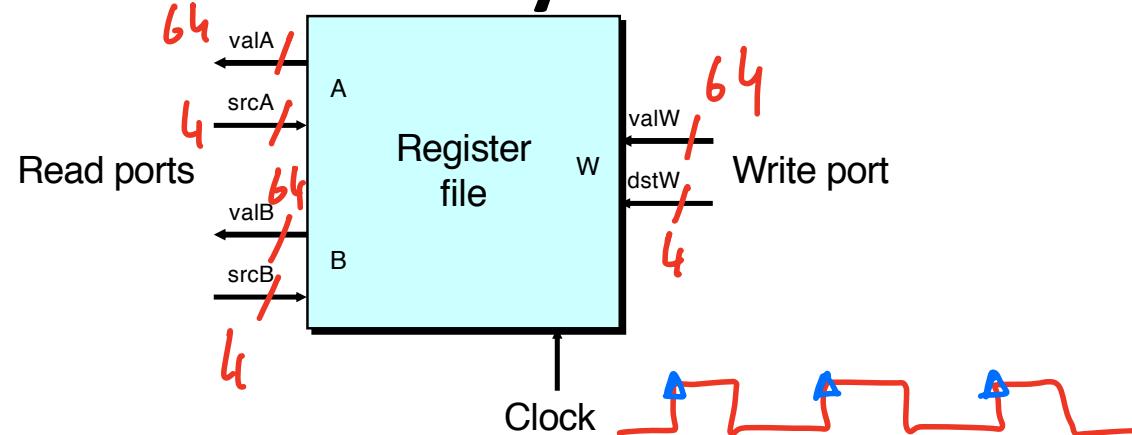
# State Machine Example



- Accumulator circuit
- Load or accumulate on each cycle

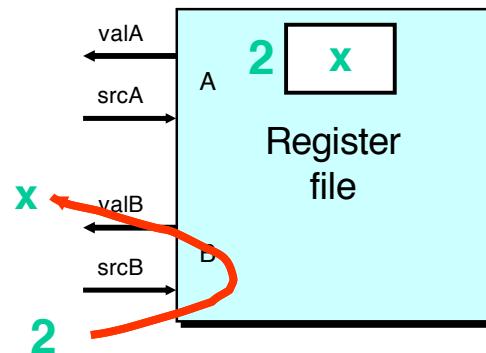


# Random-Access Memory

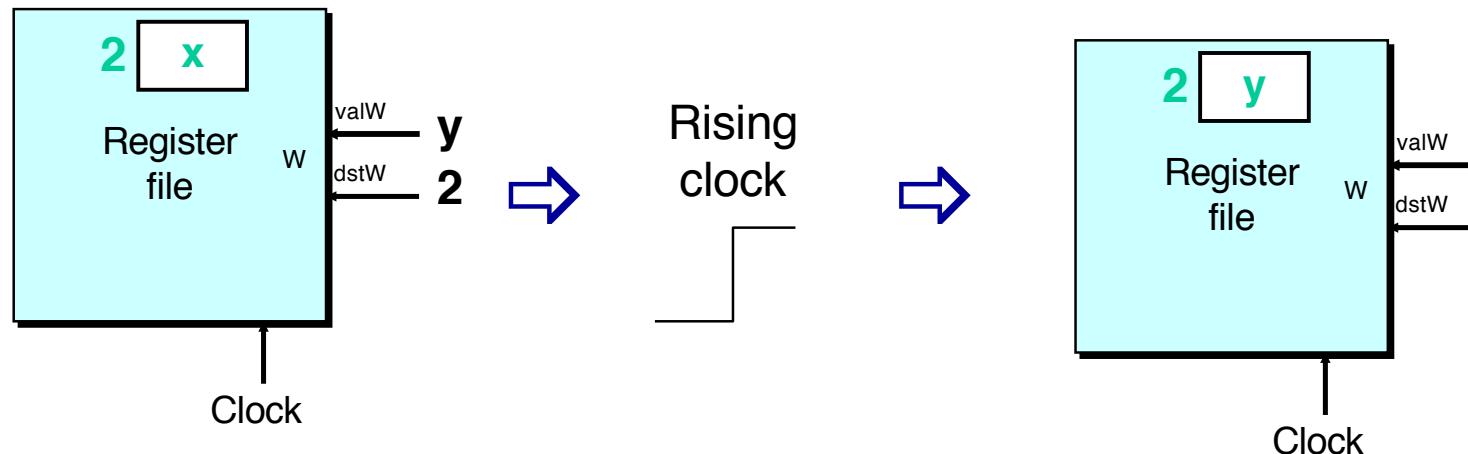


- Stores multiple words of memory
  - Address input specifies which word to read or write
- Register file
  - Holds values of program registers
  - `%rax`, `%rsp`, etc.
  - Register identifier serves as address
    - ID 15 (0xF) implies no read or write performed
- Multiple Ports
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output

# Register File Timing



- **Reading**
  - Like combinational logic
  - Output data generated based on input address
    - After some delay
  
- **Writing**
  - Like register
  - Update only as clock rises



# Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

## ■ Data Types

- `bool`: Boolean
  - `a, b, c, ...`
- `int`: words
  - `A, B, C, ...`
  - Does not specify word size---bytes, 64-bit words, ...

## ■ Statements

- `bool a = bool-expr ;`
- `int A = int-expr ;`

# HCL Operations

- Classify by type of value returned

## ■ Boolean Expressions

- Logic Operations

- $a \ \&\ b, a \ | \ b, !a$

- Word Comparisons

- $A == B, A != B, A < B, A \leq B, A \geq B, A > B$

- Set Membership

- $A \text{ in } \{ B, C, D \}$

- Same as  $A == B \ | \ A == C \ | \ A == D$

## ■ Word Expressions

- Case expressions

- $[ a : A; b : B; c : C ]$

- Evaluate test expressions  $a, b, c, \dots$  in sequence

- Return word expression  $A, B, C, \dots$  for first successful test

# Summary

## ■ Computation

- Performed by combinational logic
- Computes Boolean functions
- Continuously reacts to input changes

## ■ Storage

- Registers
  - Hold single words
  - Loaded as clock rises
- Random-access memories
  - Hold multiple words
  - Possible multiple read or write ports
  - Read word when address input changes
  - Write word as clock rises