

Processor Architecture: ISA-Y86/MIPS

1
e.g. ARM

Simplified version of
x86-64

Course Outline

■ Background

- Instruction sets
- Logic design

— HCL

■ Sequential Implementation — SEQ

- A simple, but not very fast processor design

■ Pipelining

- Get more things running simultaneously

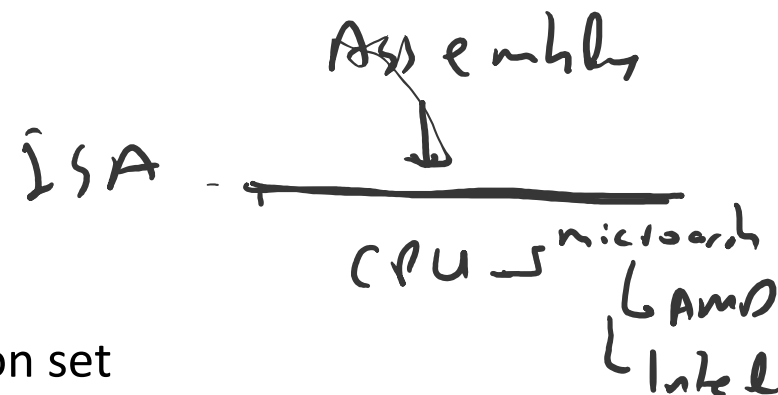
■ Pipelined Implementation — PIPE

- 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 – HCL
 - Can extend and modify
 - Test via simulation
 - Route to design using Verilog Hardware Description Language
 - See Web aside ARCH:VLOG

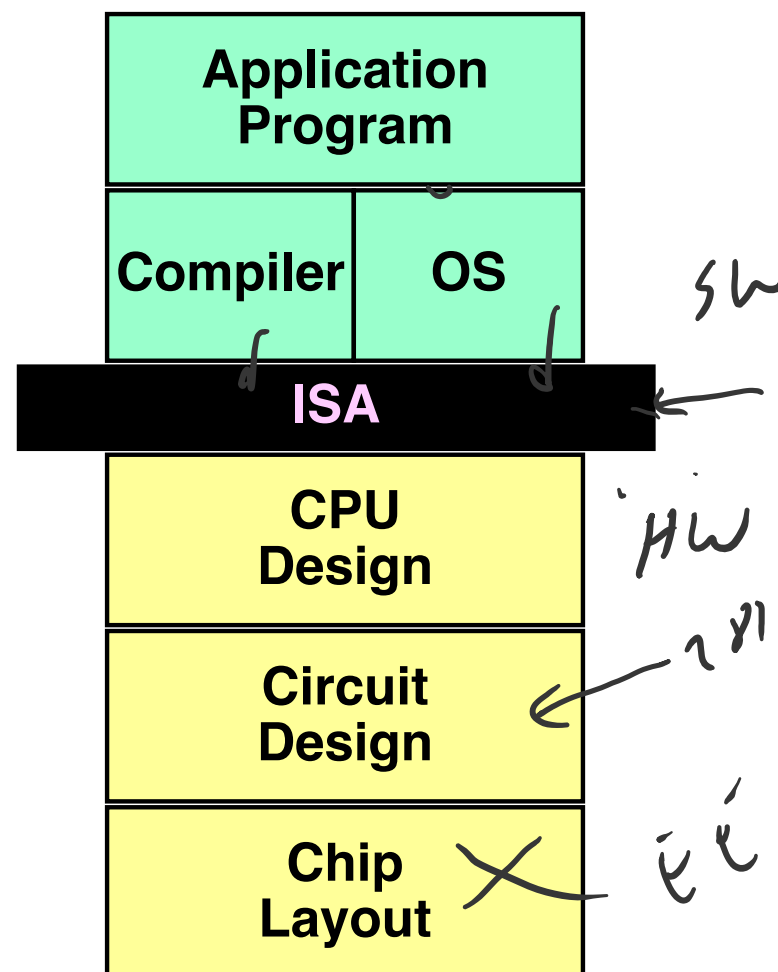
Instruction Set Architecture

■ Assembly Language View

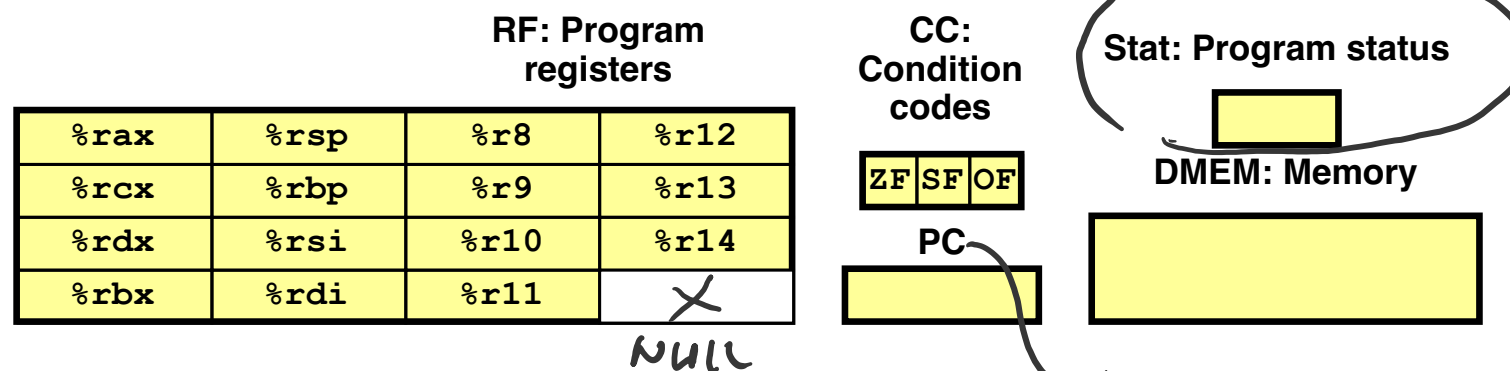
- Processor state
 - Registers, memory, ...
- Instructions
 - `addq, pushq, ret, ...`
 - How instructions are encoded as bytes

■ 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



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
 - 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

%rip ~ instruction pointer

Y86-64 Instruction Set #1

Byte

halt

nop

cmovXX rA, rB

irmovq V, rB

rmmovq rA, D(rB)

rmrmovq D(rB), rA

OPq rA, rB

jXX Dest

call Dest

ret

pushq rA

popq rA

0 1 2 3 4 5 6 7 8 9

0 0 0 0 0 0 0 0

0 0

1 0

2 fn rA rB

3 0 F rB

4 0 rA rB

5 0 rA rB

6 fn rA rB

7 fn

8 0

9 0

A 0 rA F

B 0 rA F

1 to 10 bytes

no register

i → R

R → M

M → R

V

D

D

Dest

Dest

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

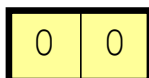
 $R \rightarrow R \equiv$ unconditional conditional

move

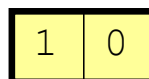
Byte

0 1 2 3 4 5 6

halt



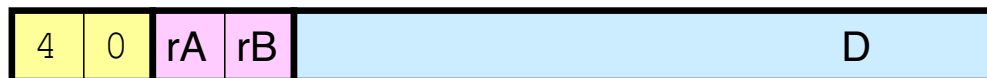
nop

cmov~~XX~~ rA, rB

irmovq V, rB



rmmovq rA, D(rB)



mrmovq D(rB), rA



OPq rA, rB



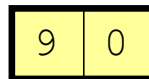
jXX Dest



call Dest



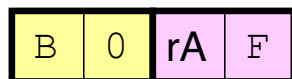
ret



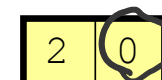
pushq rA



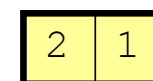
popq rA



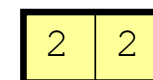
rrmovq



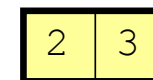
cmovle



cmovl



cmove



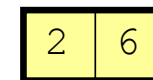
cmovne



cmovge



cmovg



Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9				
halt	0	0												
nop	1	0												
cmovXX 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	<div><div>addq</div><div>subq</div><div>andq</div><div>xorq</div></div> <div><div>6</div><div>0</div><div>1</div><div>2</div><div>3</div></div>									
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	7 0
nop	1	0								jle	7 1
cmovXX rA, rB	2	fn	rA	rB						j1	7 2
irmovq V, rB	3	0	F	rB	V					je	7 3
rmmovq rA, D(rB)	4	0	rA	rB	D					jne	7 4
mrmovq D(rB), rA	5	0	rA	rB	D					jge	7 5
OPq rA, rB	6	fn	rA	rB	8 byte address				jg	7 6	
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							

Encoding Registers

- Each register has **4-bit ID**

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	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

x86-64

↳ y86-64 - Instruction set

- Minimal
- Simpler Memory Arguments
- Split movq
- 15 register

↓
Encoding set

↳ 1-10 bytes

ret

9	0
---	---

rm movq %rax 0x8(%rbx)
 ↑ ↓ ↑
 rA D rB
 4 bits 8 byte 4 bits

 rA rB

4	0	0	3	0x8
---	---	---	---	-----

~~C → y86-64~~

C ^{gcc} → x86-64 ^{translate} → y86-64

Small
Socul

Add a new encoding → modify the architecture such that it works with new encoding

ISA

↳ CISC - x86

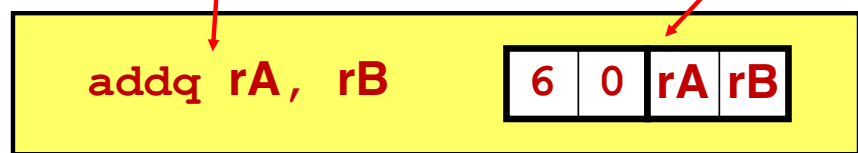
↳ RISC - Arm
 - MIPS

Instruction Example

■ Addition Instruction

Generic Form

Encoded Representation



- 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. `addq %rax, %rsi` Encoding: `60 06`
- Two-byte encoding
 - First indicates instruction type
 - Second gives source and destination registers

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

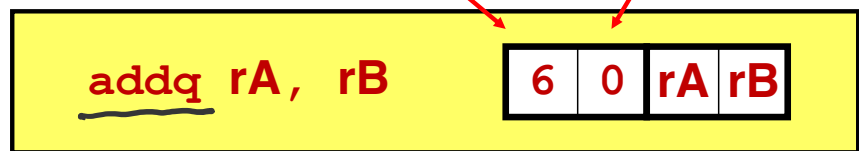
addq	6	0
subq	6	1
andq	6	2
xorq	6	3

Arithmetic and Logical Operations

Instruction Code

Function Code

Add



Subtract (rA from rB)



And



Exclusive-Or



icode *ifun*

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

Move Operations

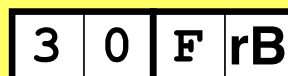
Register → Register

rmovq rA, rB



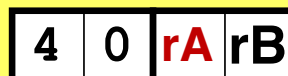
Immediate → Register

irmovq V, rB



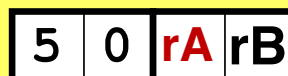
Register → Memory

rmmovq rA, D(rB)



Memory → Register

mrmmovq D(rB), rA



- 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: 30 F2 cd ab 00 00 00 00 00 00

`movq %rsp, %rbx`

Encoding: 20 43

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

Encoding: 50 15 f4 ff ff ff ff ff ff ff

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

Encoding: 40 64 1c 04 00 00 00 00 00 00

Y86-64

`irmovq $0xabcd, %rdx`

Encoding: 30 F2 cd ab 00 00 00 00 00 00

`rrmovq %rsp, %rbx`

Encoding: 20 43

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

Encoding: 50 15 f4 ff ff ff ff ff ff ff

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

0x abcd
/ little endian

-12

Conditional Move Instructions

Move Unconditionally

`rrmovq rA, rB`



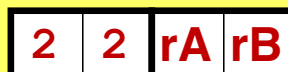
Move When Less or Equal

`cmovle rA, rB`



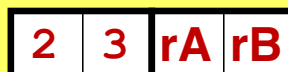
Move When Less

`cmovl rA, rB`



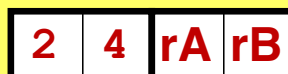
Move When Equal

`cmove rA, rB`



Move When Not Equal

`cmovne rA, rB`



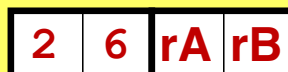
Move When Greater or Equal

`cmovge rA, rB`



Move When Greater

`cmovg rA, rB`



- 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)



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

%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
-----------------	---	---	------

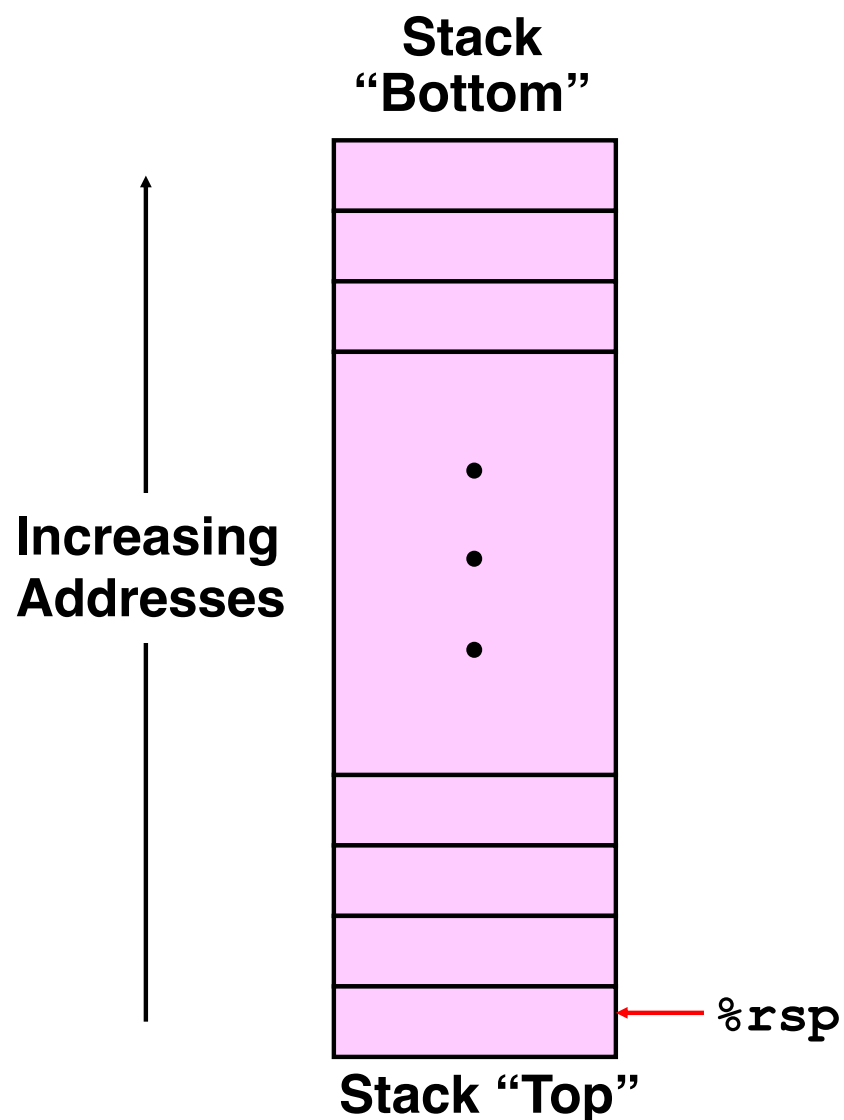
Jump When Greater or Equal

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

pushq rA



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

popq rA



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

Subroutine Call and Return

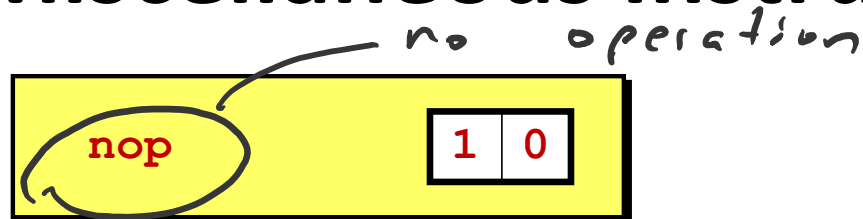


- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64



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

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 64

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

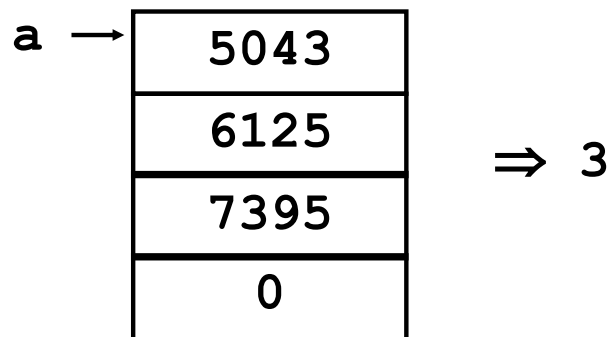
■ Try to Use C Compiler as Much as Possible

- 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;
}
```

- Compile with `gcc -Og -S`

■ 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
```

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

```
unix> yas len.y
```

text code in Y86

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

```

0x054:                | len:
0x054: 30f8010000000000000000 |    irmovq $1, %r8           # Constant 1
0x05e: 30f9080000000000000000 |    irmovq $8, %r9           # Constant 8
0x068: 30f0000000000000000000 |    irmovq $0, %rax          # len = 0
0x072: 5027000000000000000000 |    mrmovq (%rdi), %rdx       # val = *a
0x07c: 6222                |    andq %rdx, %rdx          # Test val
0x07e: 73a0000000000000000000 |    je Done                  # If zero, goto Done
0x087:                | Loop:
0x087: 6080                |    addq %r8, %rax            # len++
0x089: 6097                |    addq %r9, %rdi            # a++
0x08b: 5027000000000000000000 |    mrmovq (%rdi), %rdx       # val = *a
0x095: 6222                |    andq %rdx, %rdx          # Test val
0x097: 7487000000000000000000 |    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

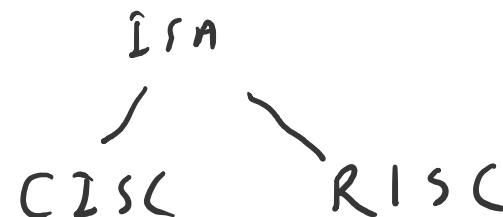
Changes to registers:

%rax:	0x0000000000000000	0x0000000000000004
%rsp:	0x0000000000000000	0x0000000000000100
%rdi:	0x0000000000000000	0x0000000000000038
%r8:	0x0000000000000000	0x0000000000000001
%r9:	0x0000000000000000	0x0000000000000008

Changes to memory:

0x00f0:	0x0000000000000000	0x0000000000000053
0x00f8:	0x0000000000000000	0x0000000000000013

CISC Instruction Sets



- Complex Instruction Set Computer
- IA32 is example – x86-64

■ 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
 - Complex address calculation

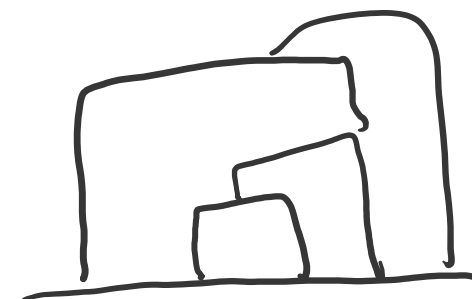
$$M[rbx + 8.rcx + 12] + = \frac{1}{5} \times$$

■ Condition codes

- Set as side effect of arithmetic and logical instructions

■ Philosophy

- Add instructions to perform “typical” programming tasks



RISC Instruction Sets

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

MIPS Registers

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

MIPS Instruction Examples

R-R

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

— register
 addu \$3, \$2, \$1 # 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: $\$3 = \$2 \ll 2$

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

*write the
result into
some other
register or
immediates
overwriting*

CISC vs. RISC

■ 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

does not produce chips
↳ just sell ISA

M1
↳ ARM

IA32 → x86-64

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

Processor Architecture: Logic Design

Overview of Logic Design

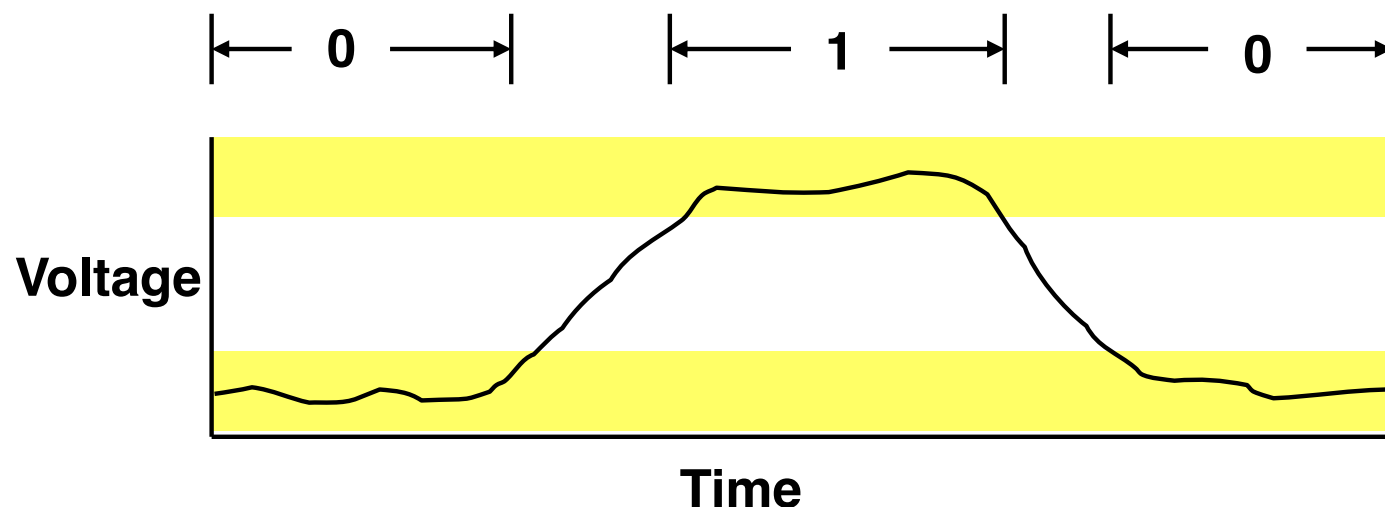
■ Fundamental Hardware Requirements

- 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

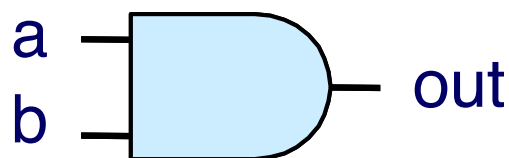
Digital Signals



- 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

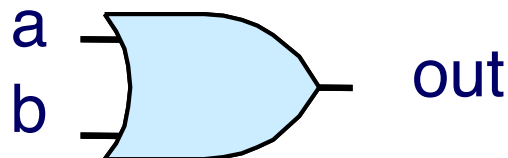
Computing with Logic Gates

And



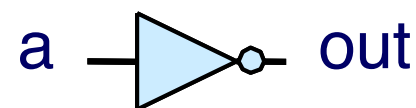
$$\text{out} = a \ \&\& \ b$$

Or



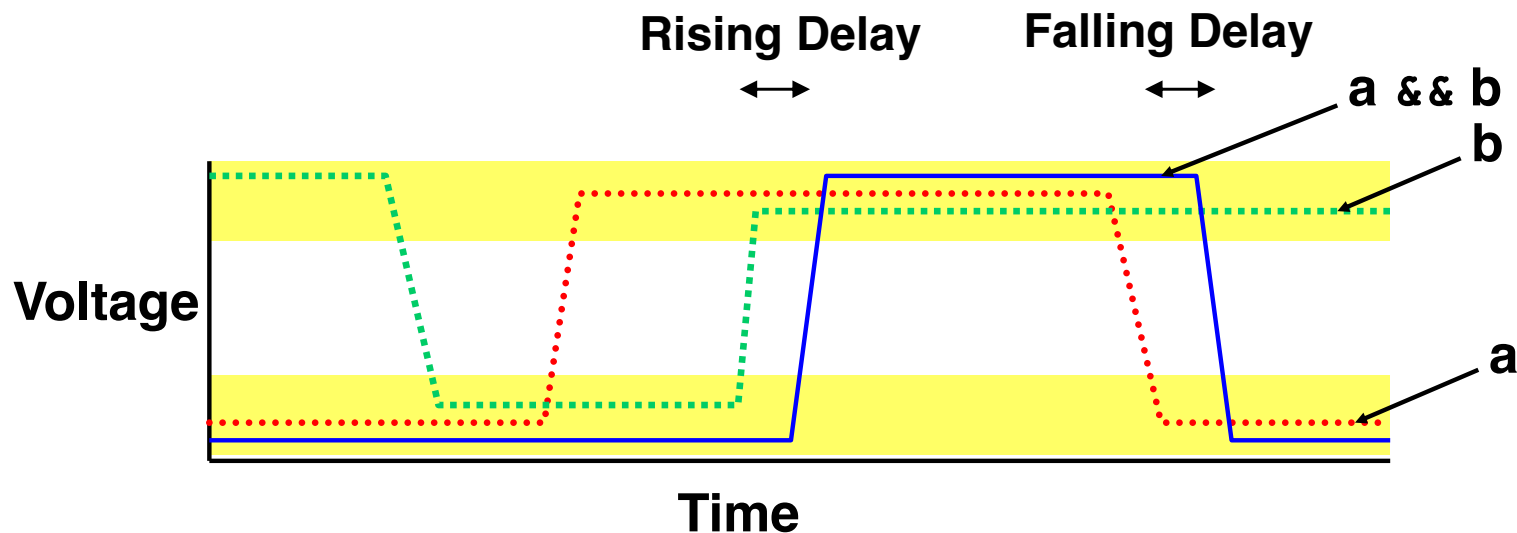
$$\text{out} = a \ || \ b$$

Not

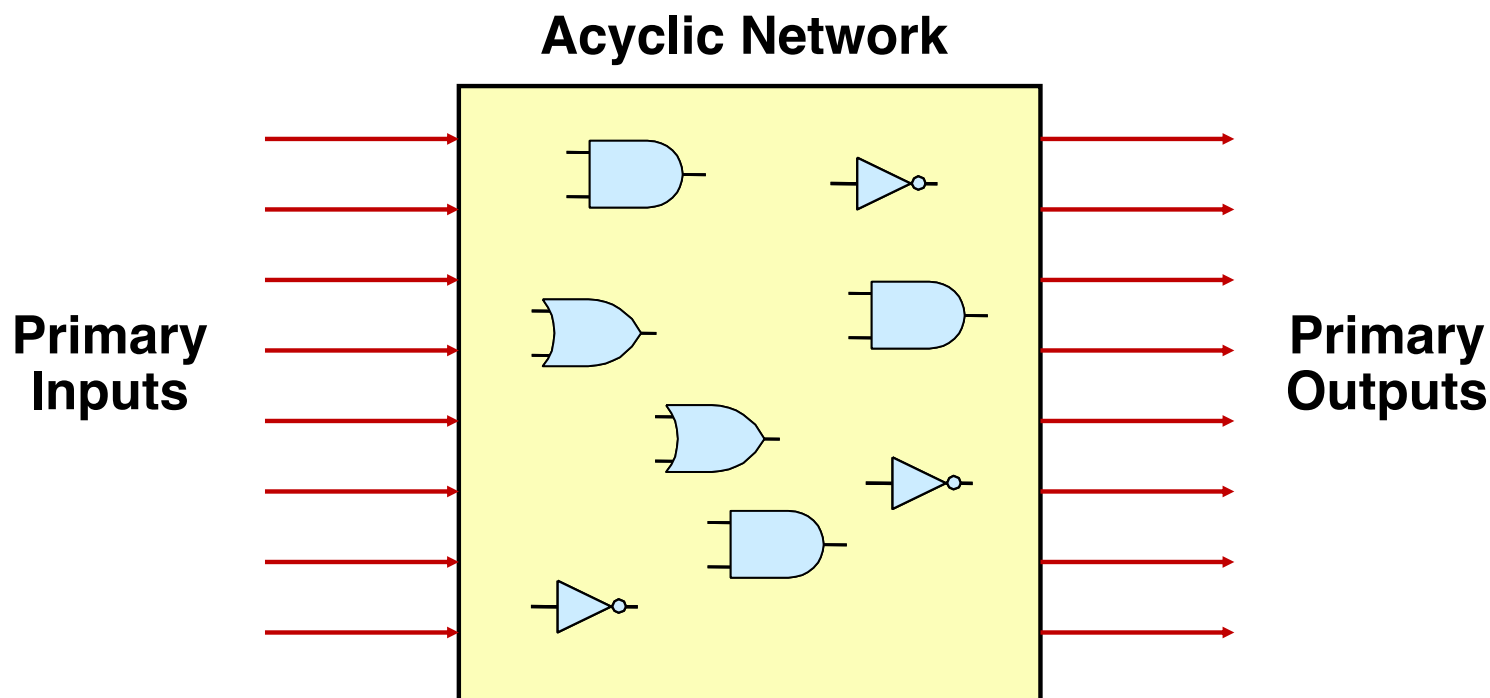


$$\text{out} = !a$$

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



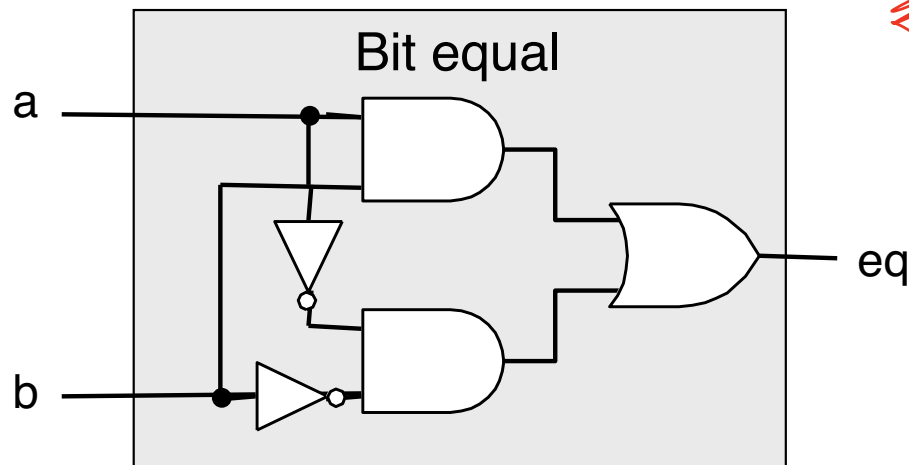
Combinational Circuits



■ 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



HCL Expression

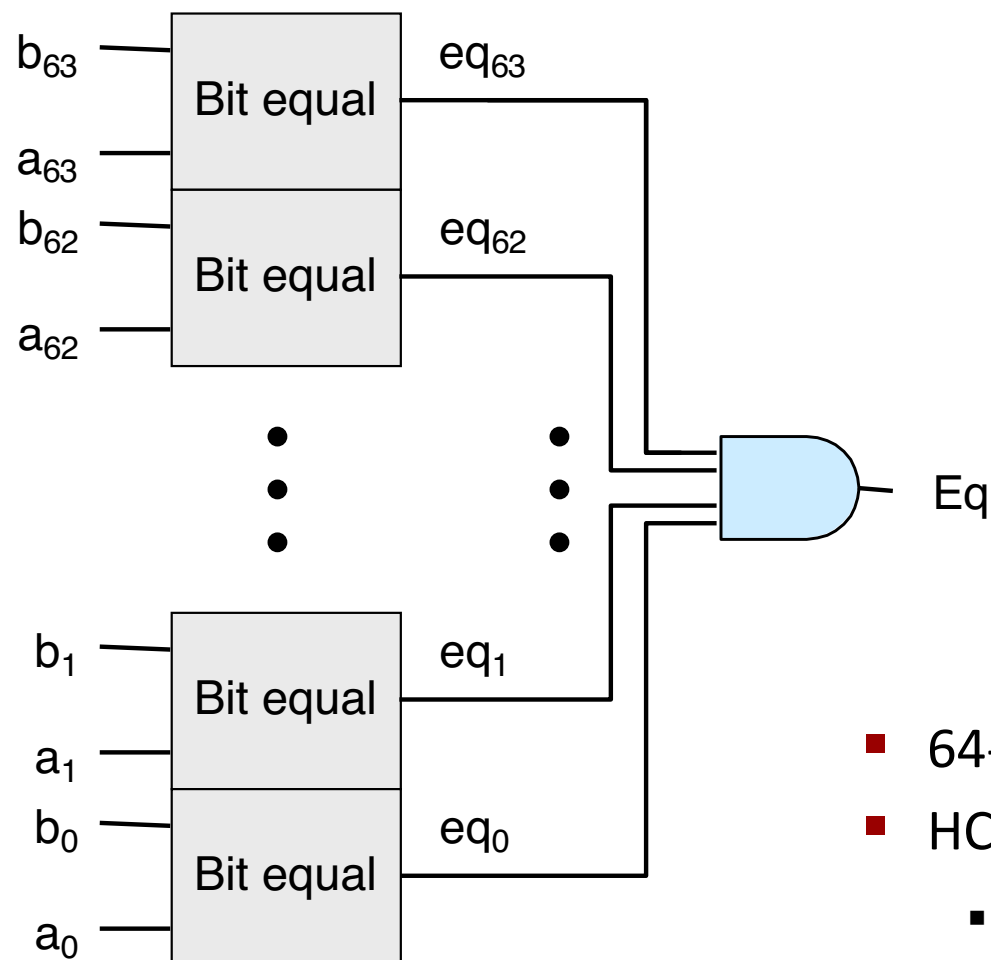
```
bool eq = (a&&b) || (!a&&!b)
```

A → word level operation

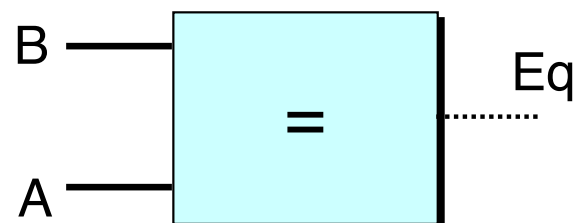
a → bit level operation

- 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

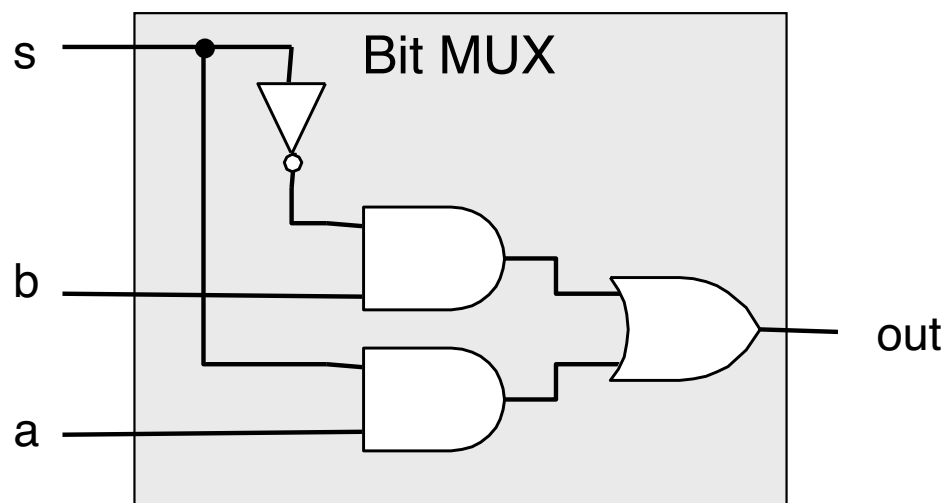


HCL Representation

`bool Eq = (A == B)`

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

Bit-Level Multiplexor

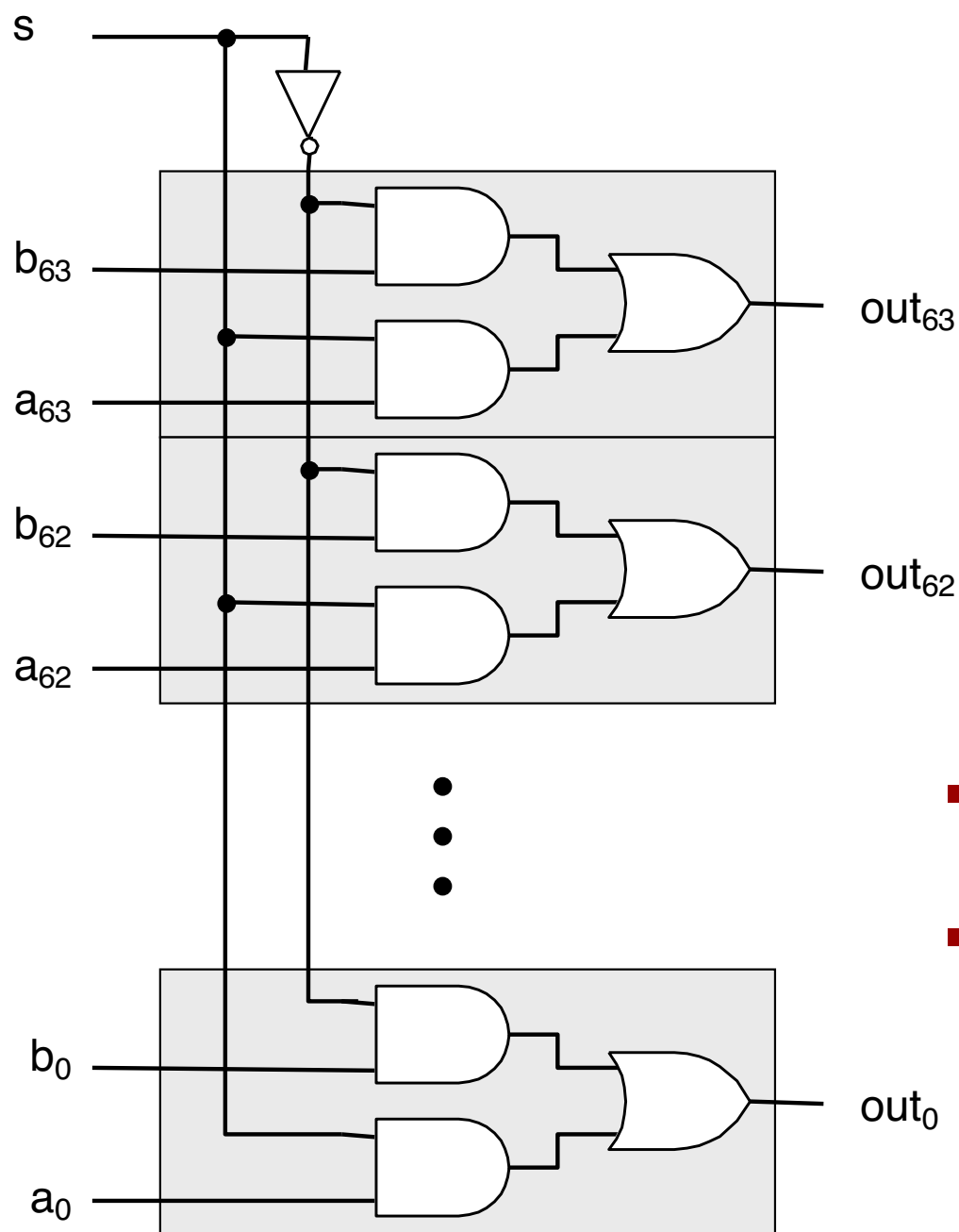


HCL Expression

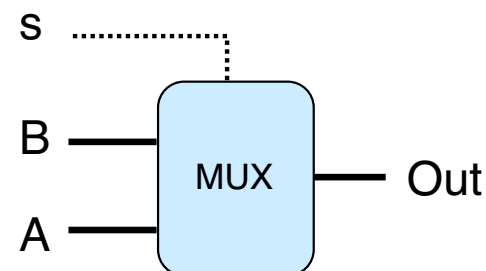
```
bool out = (s&&a) || (!s&&b)
```

- 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;
];
```

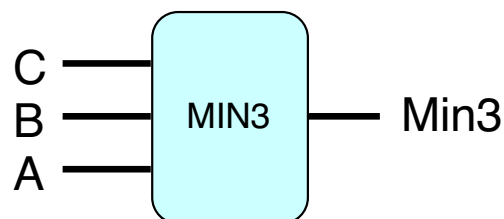
top to bottom evaluate first one the ms=1

if s==1 output=A else output=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

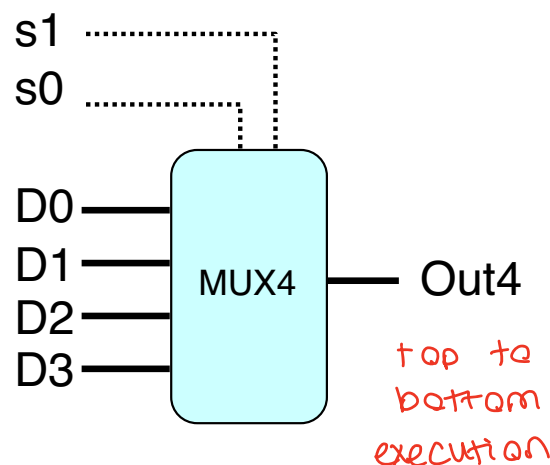
Minimum of 3 Words



```
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

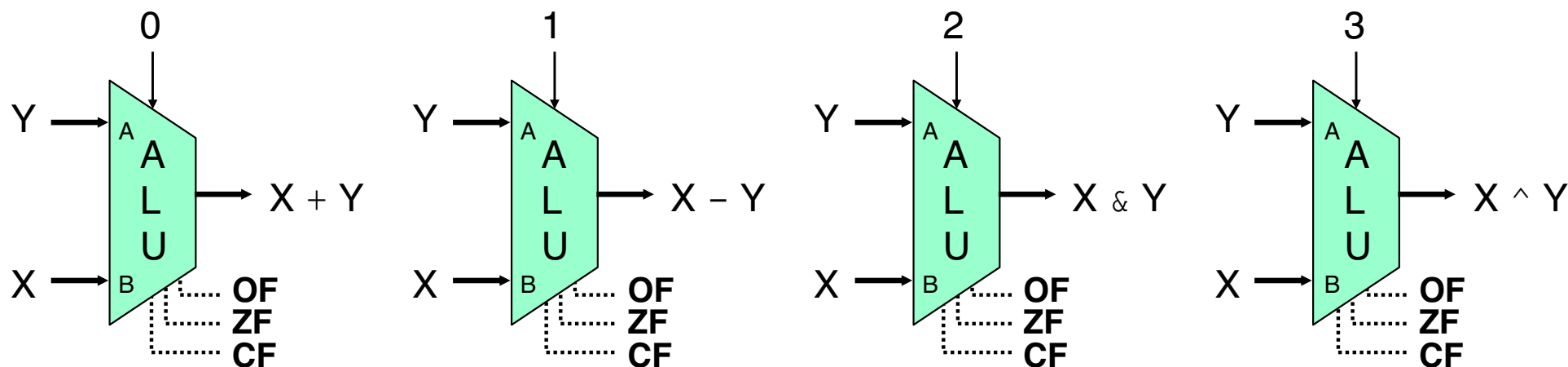
4-Way Multiplexor



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

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

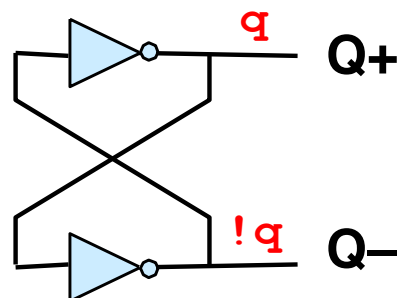
Arithmetic Logic Unit



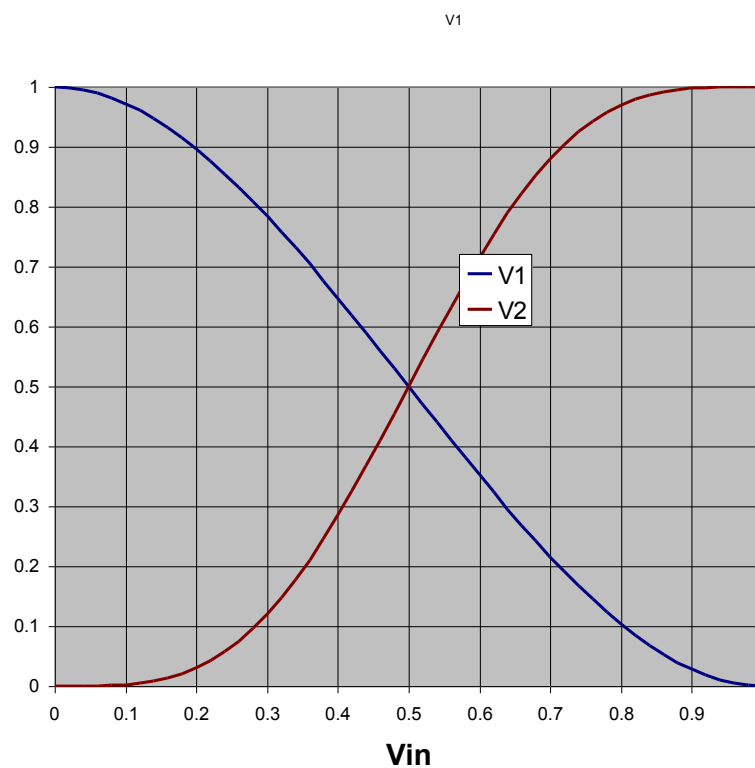
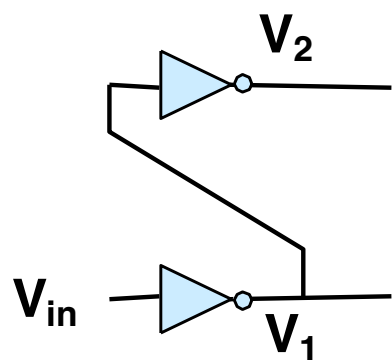
- Combinational logic
 - Continuously responding to inputs
- Control signal selects function computed
 - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

Storing 1 Bit

Bistable Element

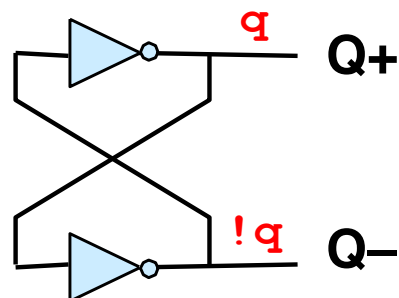


$q = 0 \text{ or } 1$

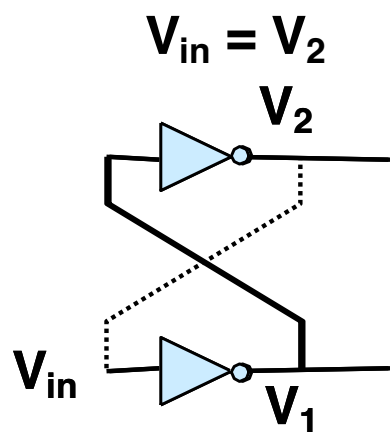


Storing 1 Bit (cont.)

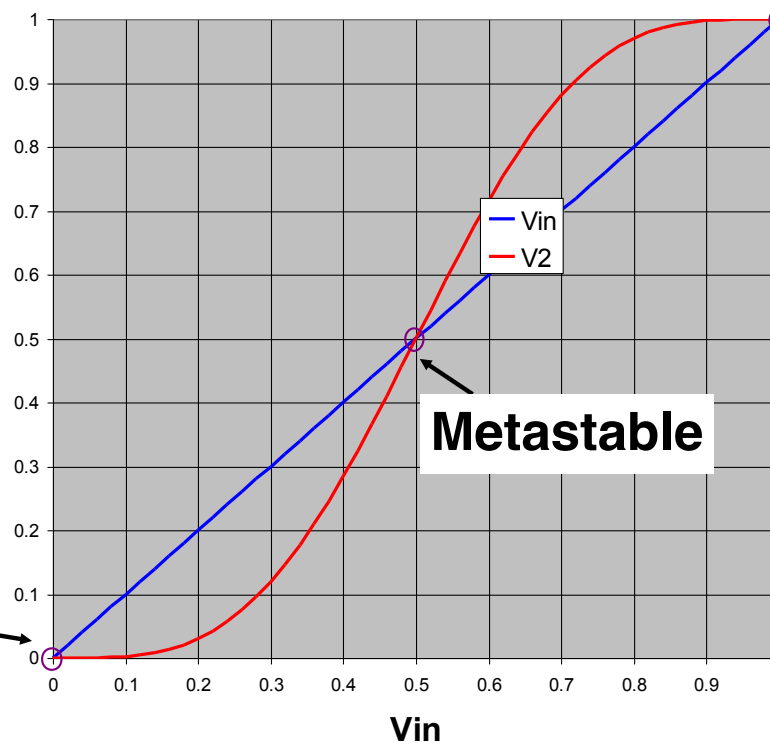
Bistable Element



$q = 0 \text{ or } 1$



Stable 0

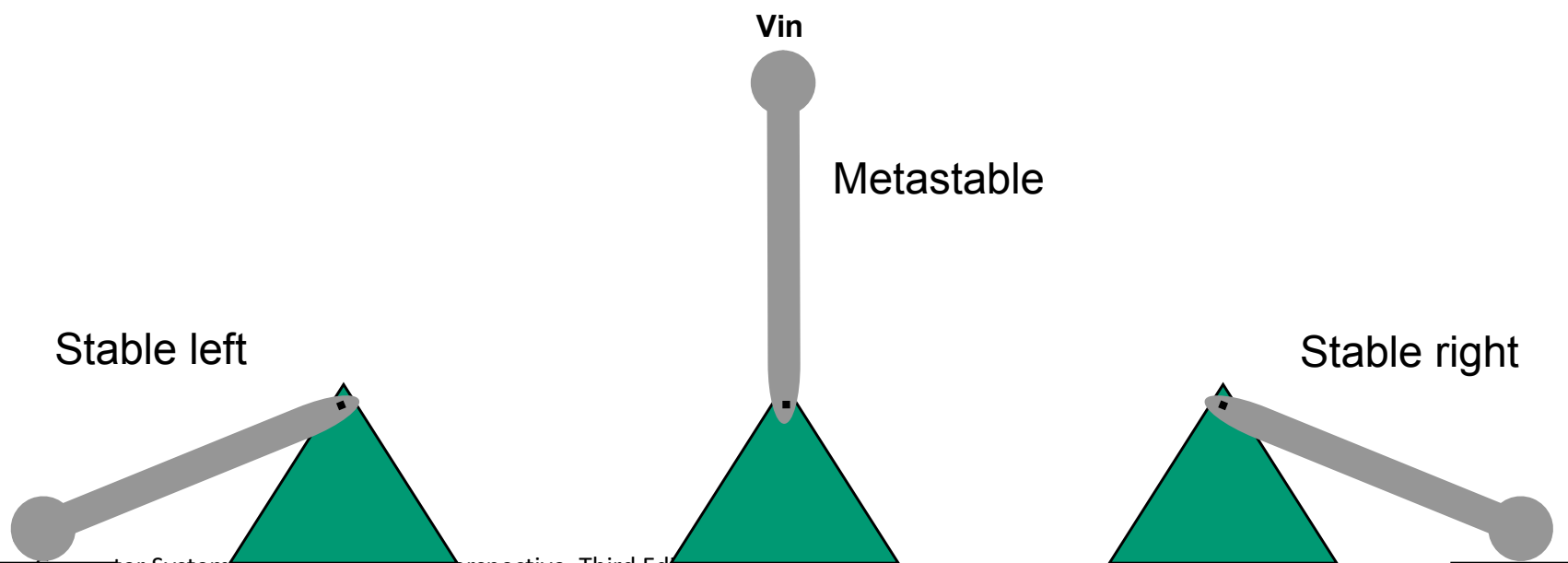
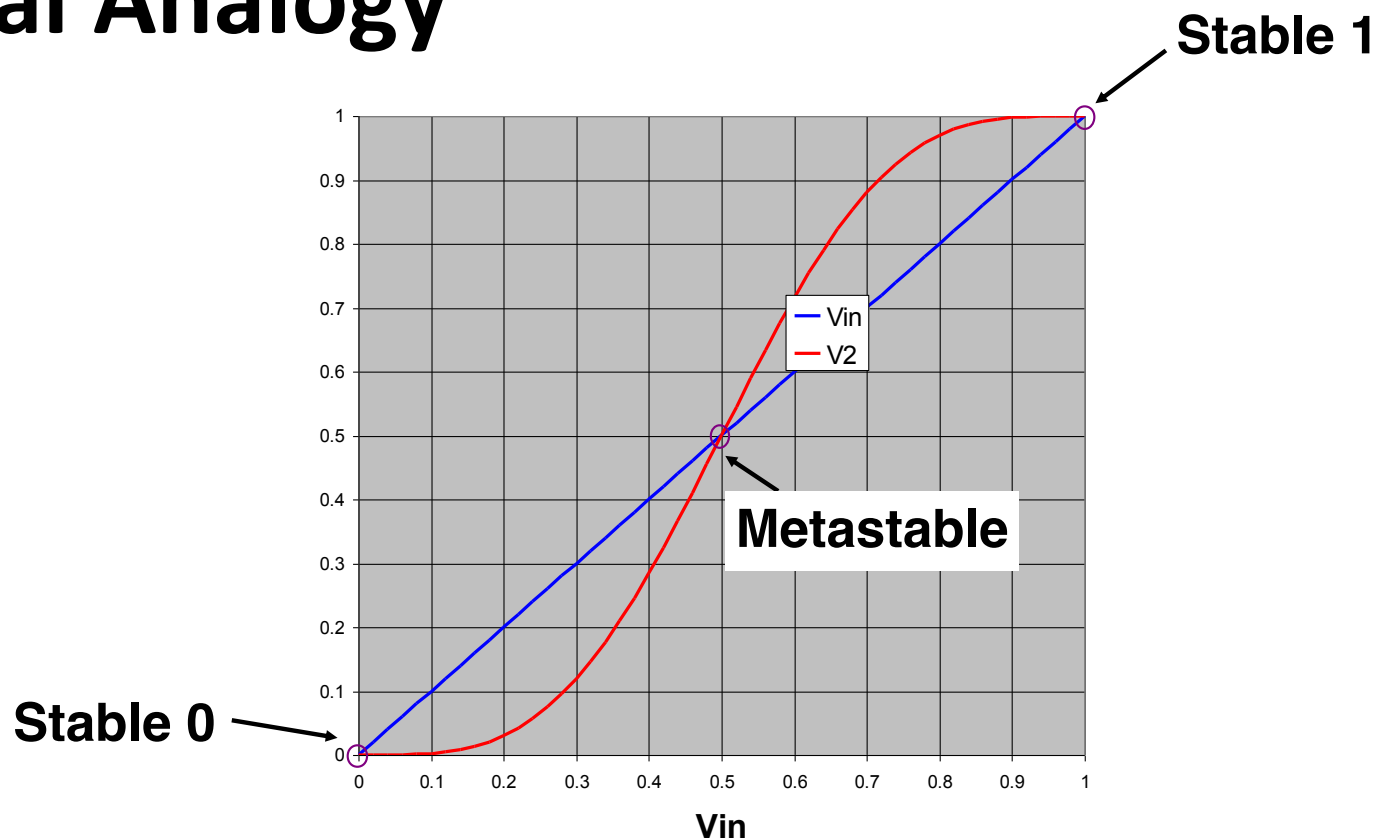


Stable 1

Metastable

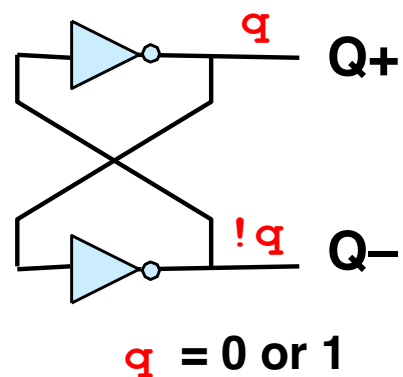
V_{in}

Physical Analogy

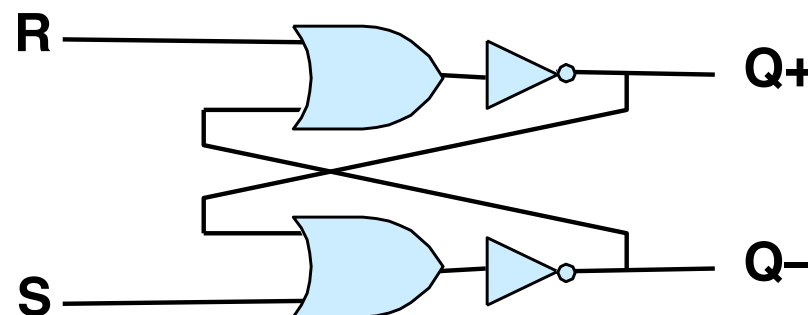


Storing and Accessing 1 Bit

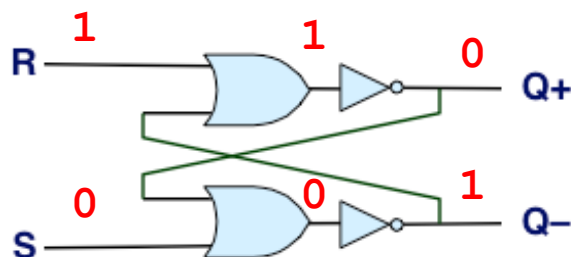
Bistable Element



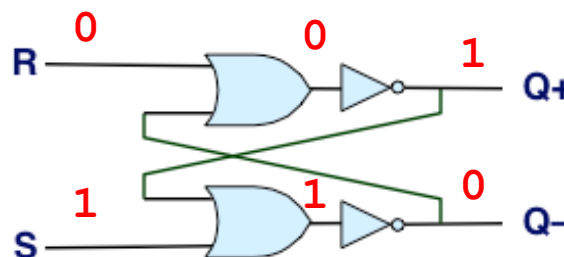
R-S Latch



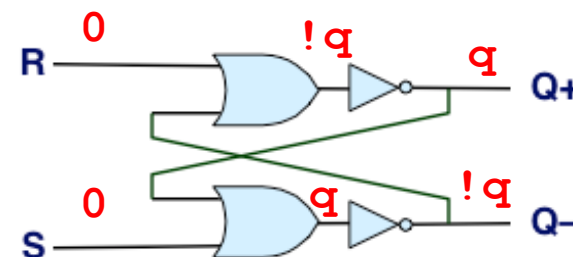
Resetting



Setting

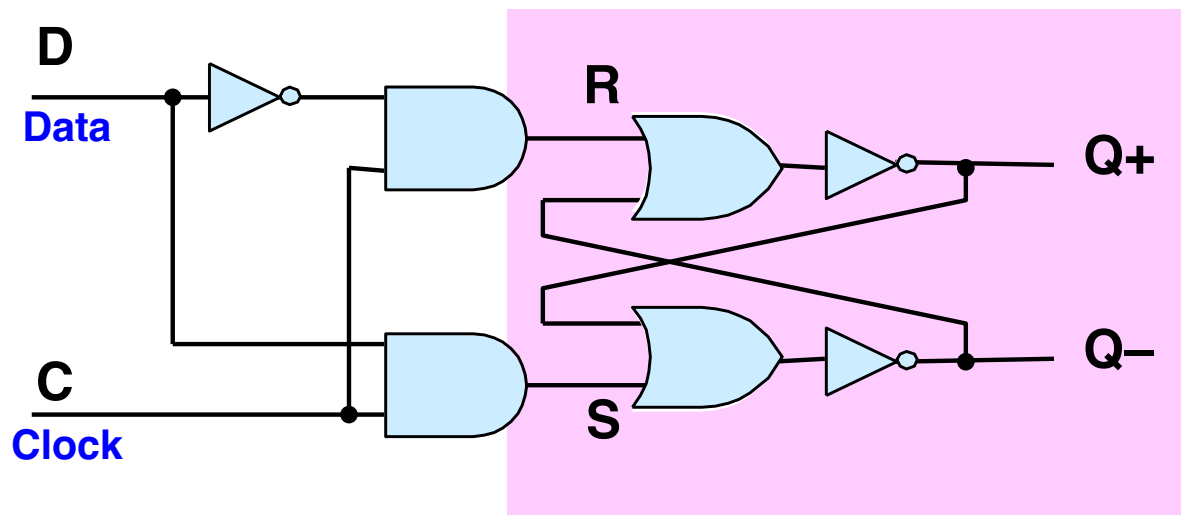


Storing

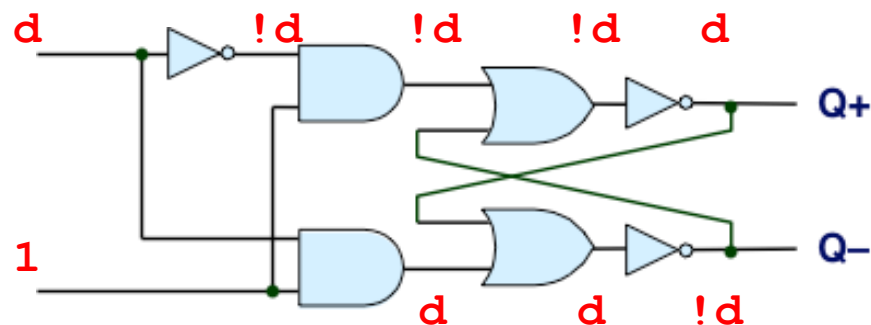


1-Bit Latch

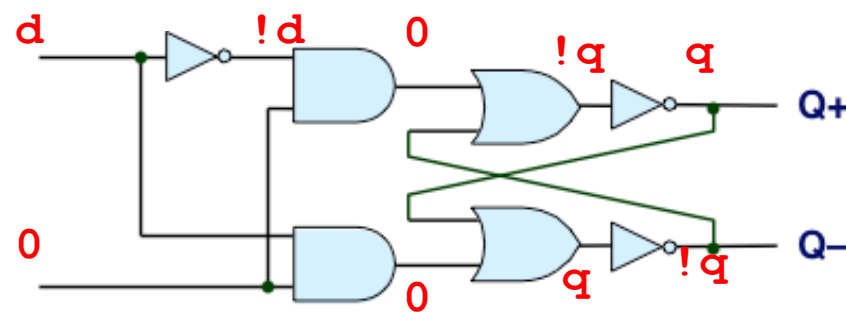
D Latch



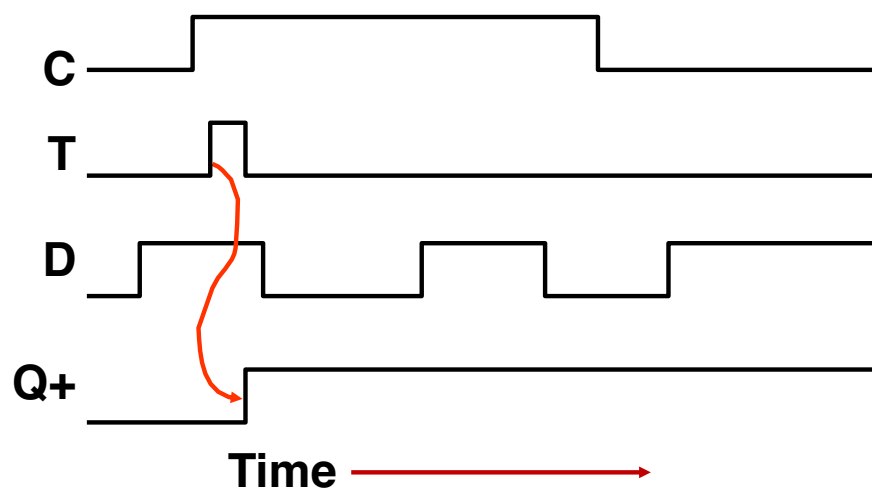
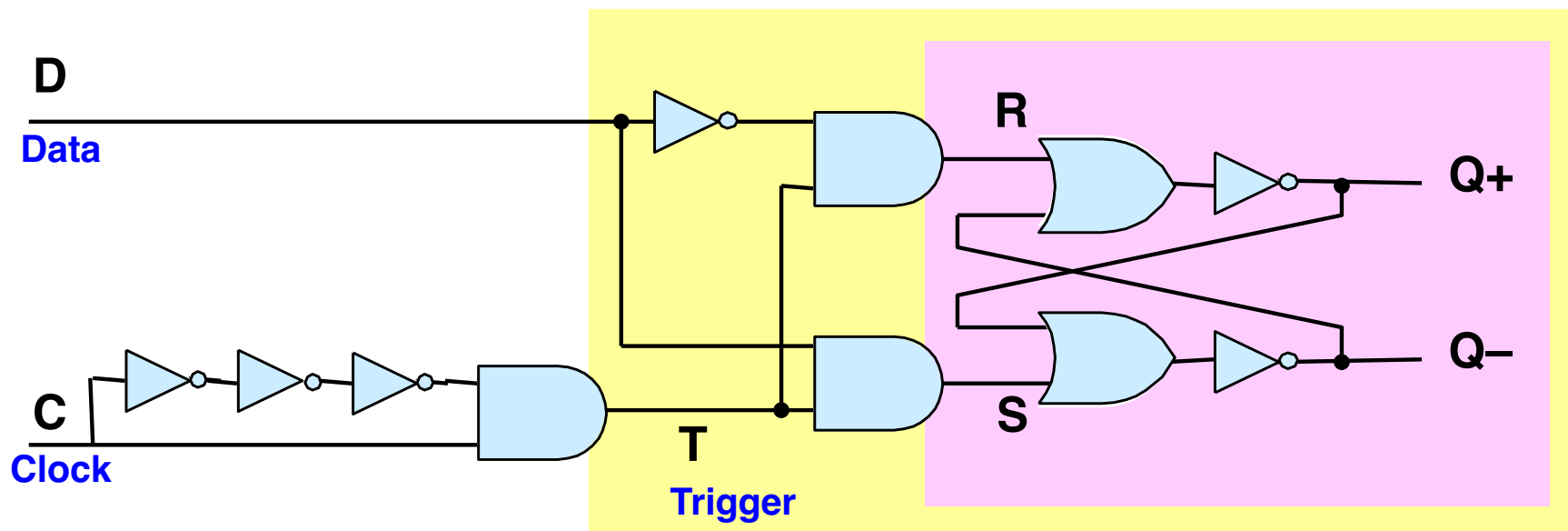
Latching



Storing

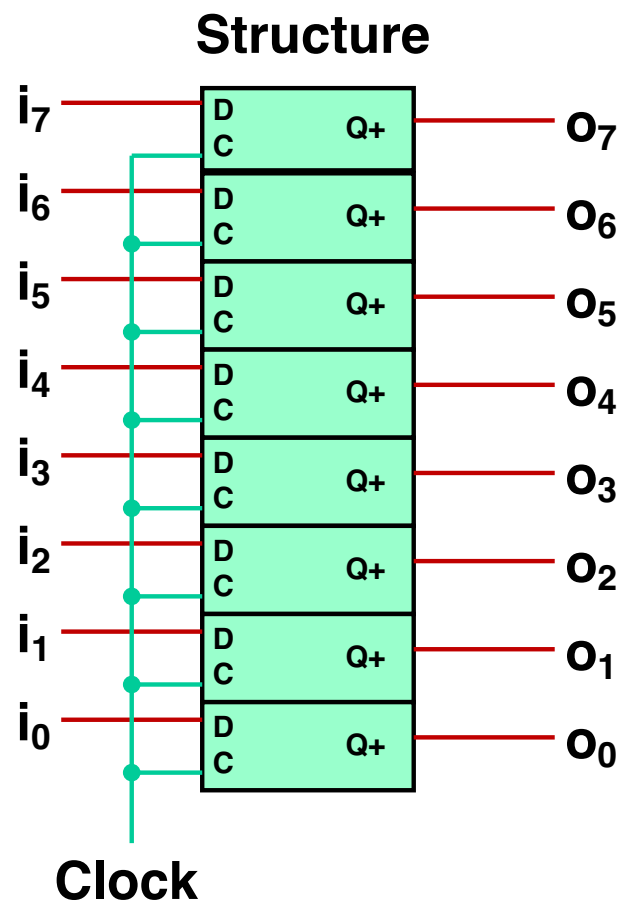


Edge-Triggered Latch



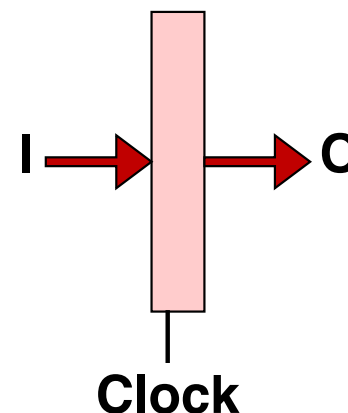
- Only in latching mode for brief period
 - Rising clock edge
- Value latched depends on data as clock rises
- Output remains stable at all other times

Registers



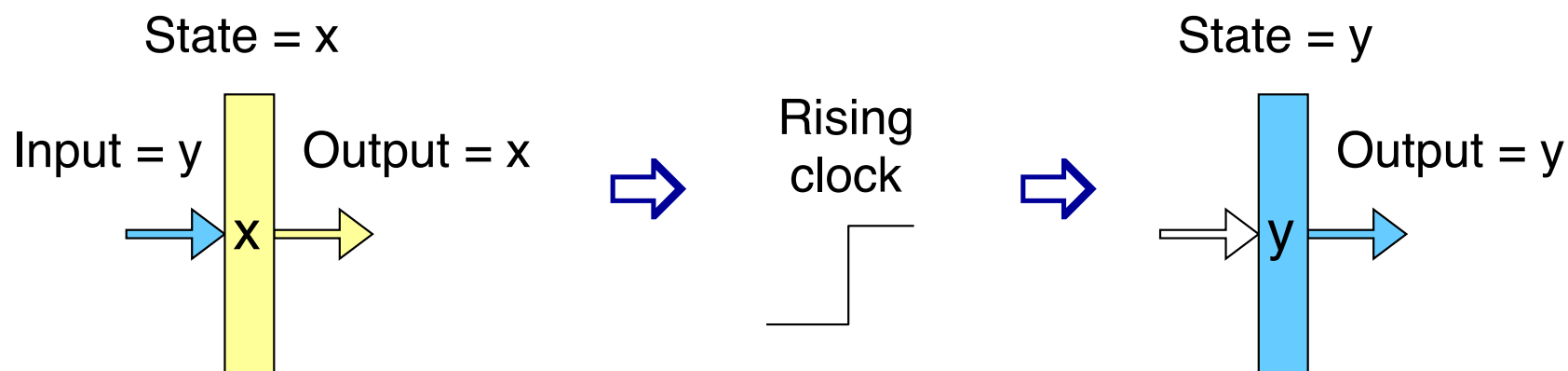
at the rising edge \rightarrow info is stored until the next rising edge

at the second rising edge \rightarrow if the signal sends sth other, store the other else don't change



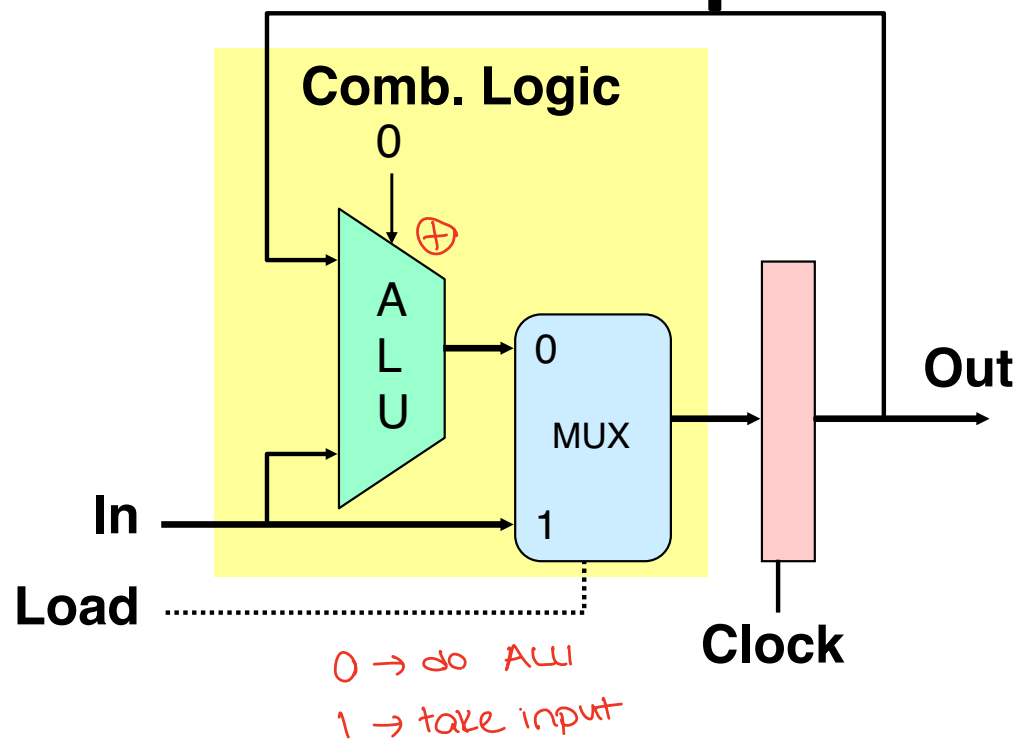
- Stores word of data
 - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

Register Operation

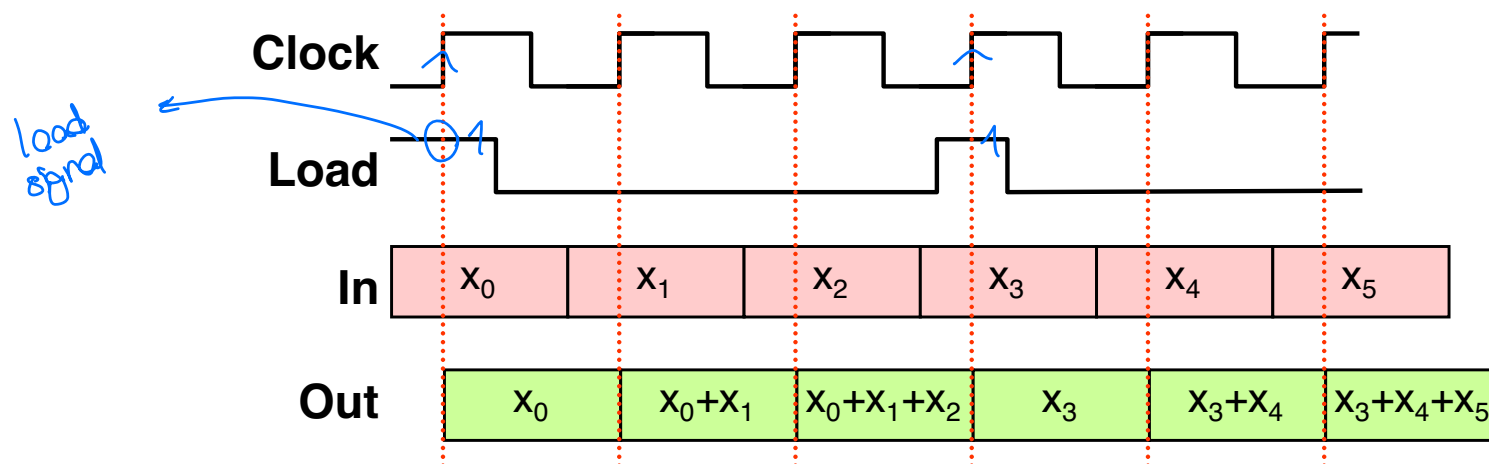


- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

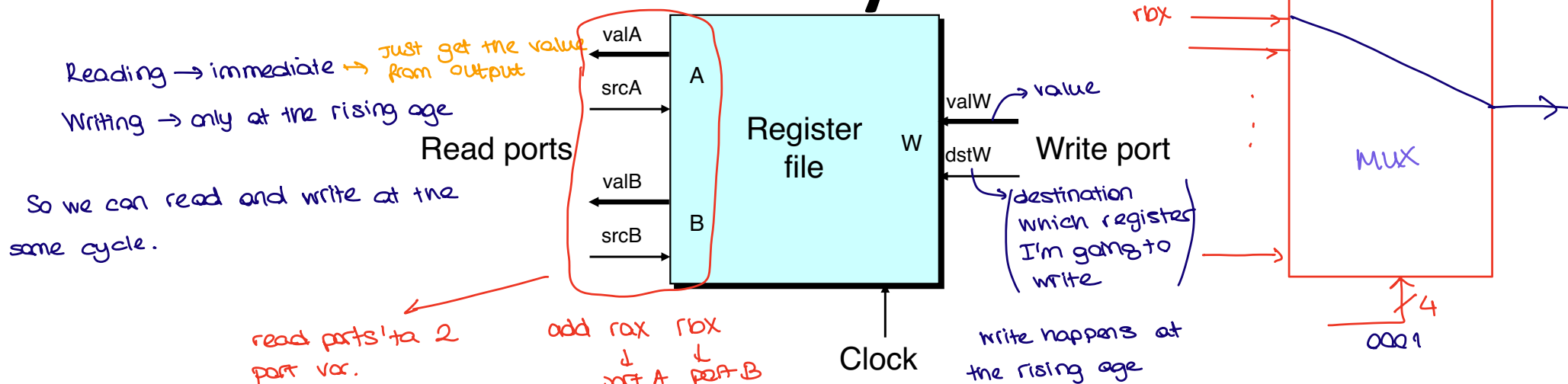
State Machine Example



- Accumulator circuit
- Load or accumulate on each cycle



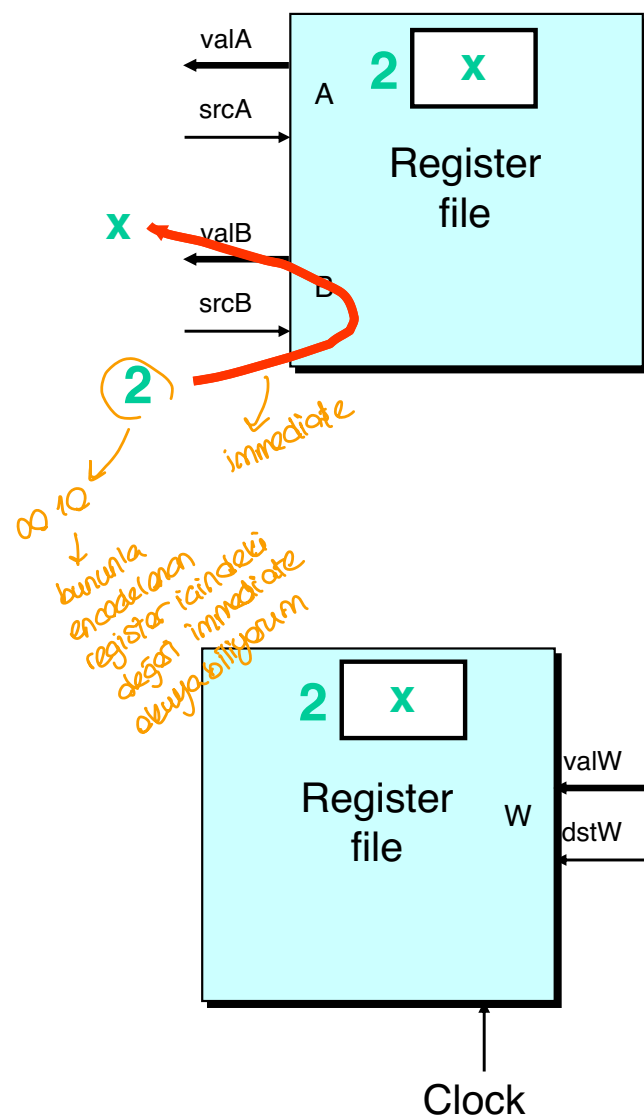
Random-Access Memory



- 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

There is binary encoding for each registers

Register File Timing

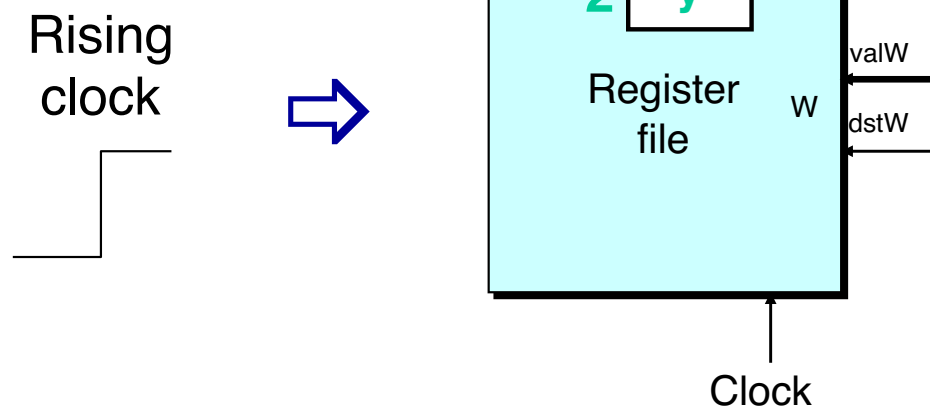


■ Reading

- 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 <= B, A >= B, A > B`
- Set Membership
 - `A 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, ...` in sequence
 - Return word expression `A, B, C, ...` 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