Processor Architecture I: ISA & Logic Design

Introduction to Computer Systems
9th Lecture, Oct 19, 2015

Instructors:

Xiangqun Chen ，Junlin Lu

Guangyu Sun ，Xuetao Guan

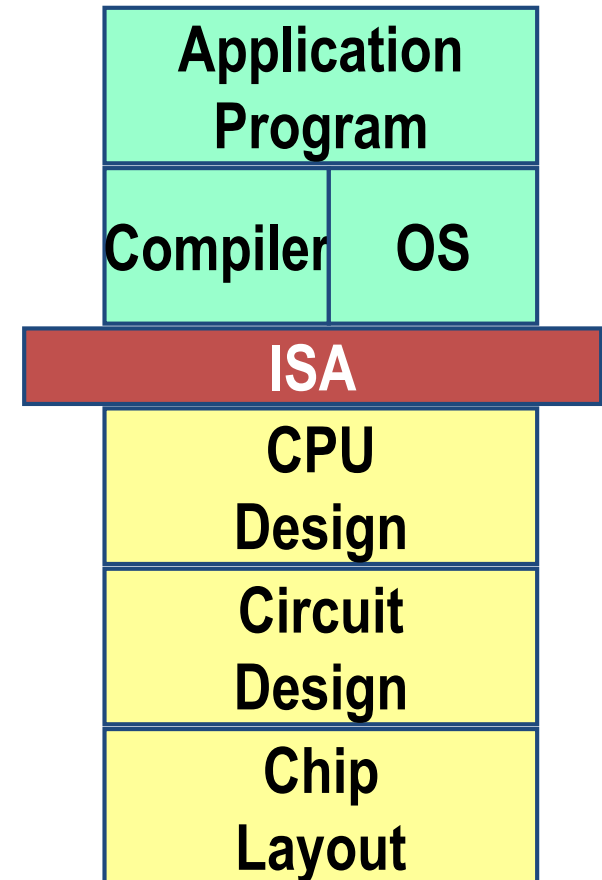# Part A

Instruction Set Architecture

Instruction Set Architecture
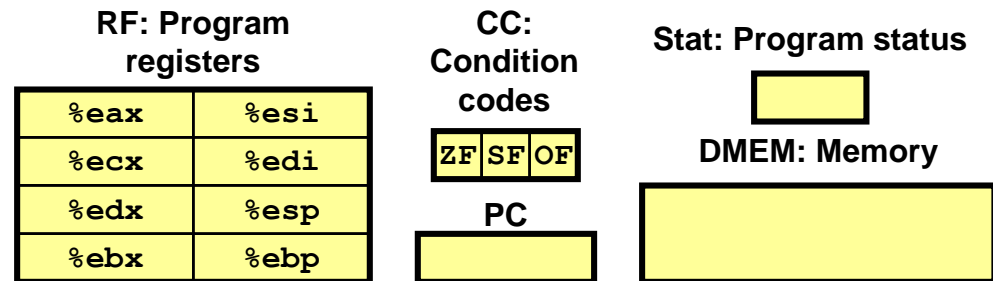
- **Assembly Language View**
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl, pushl, ret, …`
    - How instructions are encoded as bytes
- **Layer of Abstraction**
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously

Application Program
Compiler | OS
ISA
CPU Design
Circuit Design
Chip Layout

Y86 Processor State

RF: Program registers		CC: Condition codes	Stat: Program status

%eax	%esi
%ecx	%edi
%edx	%esp
%ebx	%ebp

ZF SF OF

PC

DMEM: Memory

- Program Registers
  - Same 8 as with IA32. Each 32 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

Y86 Instruction Set #1

Byte 0   1   2   3   4   5

halt `0` `0`

nop `1` `0`

cmovXX rA, rB `2` `fn` `rA` `rB`

irmovl V, rB `3` `0` `F` `rB` `V`

rmmovl rA, D(rB) `4` `0` `rA` `rB` `D`

mrmovl D(rB), rA `5` `0` `rA` `rB` `D`

OPl rA, rB `6` `fn` `rA` `rB`

jXX Dest `7` `fn` `Dest`

call Dest `8` `0` `Dest`

ret `9` `0`

pushl rA `A` `0` `rA` `F`

popl rA `B` `0` `rA` `F`

Y86 Instructions

- **Format**
    - 1–6 bytes of information read from memory
        - Can determine instruction length from first byte
        - Not as many instruction types, and simpler encoding than with IA32
    - Each accesses and modifies some part(s) of the program state

Y86 Instruction Set #2

Byte   0   1   2   3   4   5

```
halt              | 0 | 0 |

nop               | 1 | 0 |

cmovXX rA, rB     | 2 | fn | rA | rB |

irmovl V, rB      | 3 | 0 | F | rB | V        |

rmmovl rA, D(rB)  | 4 | 0 | rA | rB | D       |

mrmovl D(rB), rA  | 5 | 0 | rA | rB | D       |

OPl rA, rB        | 6 | fn | rA | rB |

jXX Dest          | 7 | fn | Dest            |

call Dest         | 8 | 0 | Dest             |

ret               | 9 | 0 |

pushl rA          | A | 0 | rA | F |

popl rA           | B | 0 | rA | F |
```

rrmovl | 2 | 0 |

cmovle | 2 | 1 |

cmovl | 2 | 2 |

cmove | 2 | 3 |

cmovne | 2 | 4 |

cmovge | 2 | 5 |

cmovg | 2 | 6 |

7

Y86 Instruction Set #3

Byte 0 1 2 3 4 5

`halt` | 0 | 0 |

`nop` | 1 | 0 |

`cmovXX rA, rB` | 2 | fn | rA | rB |

`irmovl V, rB` | 3 | 0 | F | rB | V |

`rmmovl rA, D(rB)` | 4 | 0 | rA | rB | D |

`mrmovl D(rB), rA` | 5 | 0 | rA | rB | D |

`OPl rA, rB` | 6 | fn | rA | rB |

`jXX Dest` | 7 | fn | Dest |

`call Dest` | 8 | 0 | Dest |

`ret` | 9 | 0 |

`pushl rA` | A | 0 | rA | F |

`popl rA` | B | 0 | rA | F |

`addl` | 6 | 0 |

`subl` | 6 | 1 |

`andl` | 6 | 2 |

`xorl` | 6 | 3 |

Y86 Instruction Set #4

Byte　　　　　0　　　1　　　2　　　3　　　4　　　5

`halt` `| 0 | 0 |`

`nop` `| 1 | 0 |`

`rrmovl rA, rB` `| 2 | fn | rA | rB |`

`irmovl V, rB` `| 3 | 0 | F | rB | V |`

`rmmovl rA, D(rB)` `| 4 | 0 | rA | rB | D |`

`mrmovl D(rB), rA` `| 5 | 0 | rA | rB | D |`

`OPl rA, rB` `| 6 | fn | rA | rB |`

`jXX Dest` `| 7 | fn | Dest |`

`call Dest` `| 8 | 0 | Dest |`

`ret` `| 9 | 0 |`

`pushl rA` `| A | 0 | rA | F |`

`popl rA` `| B | 0 | rA | F |`

`jmp` `| 7 | 0 |`

`jle` `| 7 | 1 |`

`jl` `| 7 | 2 |`

`je` `| 7 | 3 |`

`jne` `| 7 | 4 |`

`jge` `| 7 | 5 |`

`jg` `| 7 | 6 |`

Encoding Registers

■ **Each register has 4-bit ID**

%eax	0		%esi	6
%ecx	1		%edi	7
%edx	2		%esp	4
%ebx	3		%ebp	5

   ▪ Same encoding as in IA32

■ **Register ID 15 (0xF) indicates "no register"**

   ▪ Will use this in our hardware design in multiple places

Instruction Example

■ **Addition Instruction**

Generic Form

Encoded Representation

```
addl rA, rB        6  0  rA  rB
```
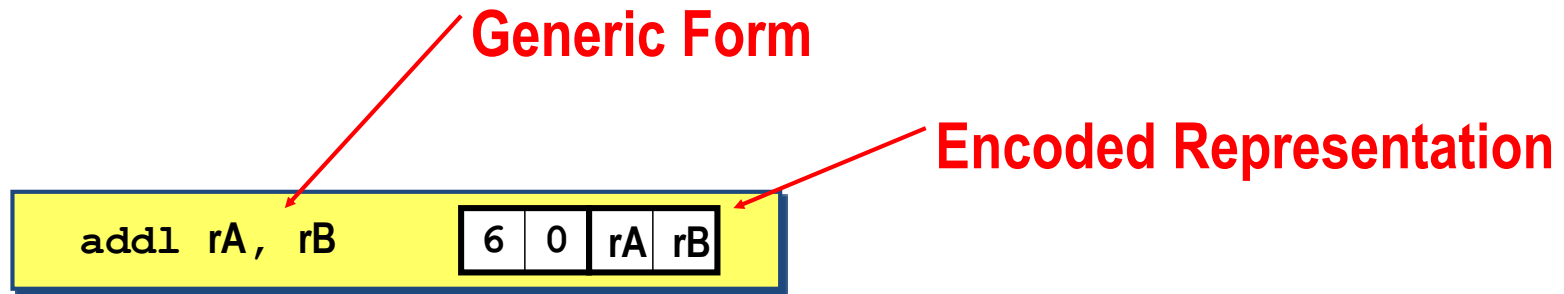
- ▪ Add value in register rA to that in register rB
  - ▪ Store result in register rB
  - ▪ Note that Y86 only allows addition to be applied to register data
- ▪ Set condition codes based on result
- ▪ e.g., **addl %eax,%esi**   Encoding: **60 06**
- ▪ Two-byte encoding
  - ▪ First indicates instruction type
  - ▪ Second gives source and destination registers

Arithmetic and Logical Operations

Instruction Code   **Function Code**

Add

| addl rA, rB | 6 | 0 | rA | rB |

Subtract (rA from rB)

| subl rA, rB | 6 | 1 | rA | rB |

And

| andl rA, rB | 6 | 2 | rA | rB |

Exclusive-Or

| xorl rA, rB | 6 | 3 | rA | rB |
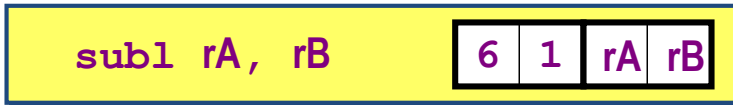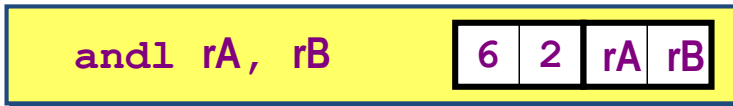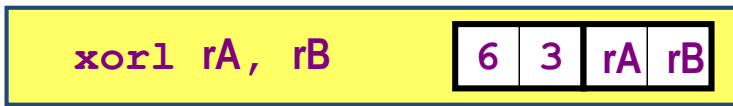
- Refer to generically as "OP1"
- Encodings differ only by "function code"
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

Move Operations

rrmovl rA, rB	2	0	rA	rB
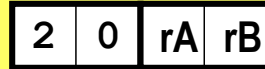
Register --> Register

irmovl V, rB	3	0	F	rB	V

Immediate --> Register

rmmovl rA, D(rB)	4	0	rA	rB	D

Register --> Memory

mrmovl D(rB), rA	5	0	rA	rB	D

Memory --> Register

- Like the IA32 `movl` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

IA32	Y86	Encoding
`movl $0xabcd, %edx`	`irmovl $0xabcd, %edx`	`30 F2 cd ab 00 00`
`movl %esp, %ebx`	`rrmovl %esp, %ebx`	`20 43`
`movl -12(%ebp),%ecx`	`mrmovl -12(%ebp),%ecx`	`50 15 f4 ff ff ff`
`movl %esi,0x41c(%esp)`	`rmmovl %esi,0x41c(%esp)`	`40 64 1c 04 00 00`

`movl $0xabcd, (%eax)`	—
`movl %eax, 12(%eax,%edx)`	—
`movl (%ebp,%eax,4),%ecx`	—

Conditional Move Instructions

Move Unconditionally

	2	0	rA	rB	
rrmovl rA, rB					

Move When Less or Equal

	2	1	rA	rB	
cmovle rA, rB					

Move When Less

	2	2	rA	rB	
cmovl rA, rB					

Move When Equal

	2	3	rA	rB	
cmove rA, rB					

Move When Not Equal

	2	4	rA	rB	
cmovne rA, rB					

Move When Greater or Equal

	2	5	rA	rB	
cmovge rA, rB					

Move When Greater

	2	6	rA	rB	
cmovg rA, rB					

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

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

- Refer to generically as "jXX"
- Encodings differ only by "function code"
- Based on values of condition codes
- Same as IA32 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in IA32

Y86 Program Stack

Stack "Bottom"

Increasing Addresses

Stack "Top"

%esp
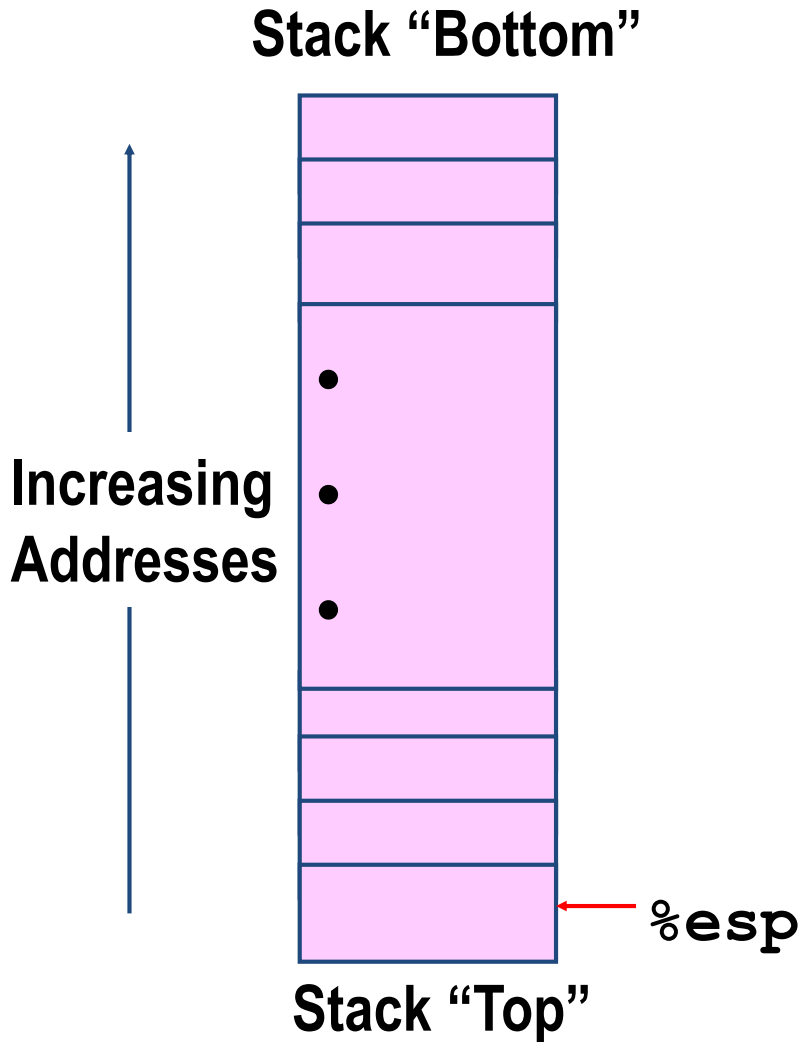
- Region of memory holding program data
- Used in Y86 (and IA32) for supporting procedure calls
- Stack top indicated by `%esp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer

Stack Operations

| `pushl rA` | | A | 0 | rA | F |

- **Decrement `%esp` by 4**

- **Store word from rA to memory at `%esp`**

- **Like IA32**

| `popl rA` | | B | 0 | rA | F |

- **Read word from memory at `%esp`**

- **Save in rA**

- **Increment `%esp` by 4**

- **Like IA32**

Subroutine Call and Return

`call Dest`	8	0	Dest

- **Push address of next instruction onto stack**
- **Start executing instructions at Dest**
- **Like IA32**

`ret`	9	0

- **Pop value from stack**
- **Use as address for next instruction**
- **Like IA32**

Miscellaneous Instructions

nop	1	0

- **Don't do anything**

halt	0	0

- **Stop executing instructions**
- **IA32 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

Mnemonic	Code
AOK	1

- **Normal operation**

Mnemonic	Code
HLT	2

- **Halt instruction encountered**

Mnemonic	Code
ADR	3

- **Bad address (either instruction or data) encountered**

Mnemonic	Code
INS	4

- **Invalid instruction encountered**

Desired Behavior

- **If AOK, keep going**
- **Otherwise, stop program execution**

Writing Y86 Code

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

- ▪ Write code in C
- ▪ Compile for IA32 with `gcc34 –O1 –S`
  - ▪ Newer versions of GCC do too much optimization
  - ▪ Use `ls /usr/bin/gcc`* to find what versions are available
- ▪ Transliterate into Y86

■ **Coding Example**

- ▪ Find number of elements in null-terminated list
  ```
  int len1(int a[]);
  ```

a →

5043
6125
7395
0

$\Rightarrow$ 3
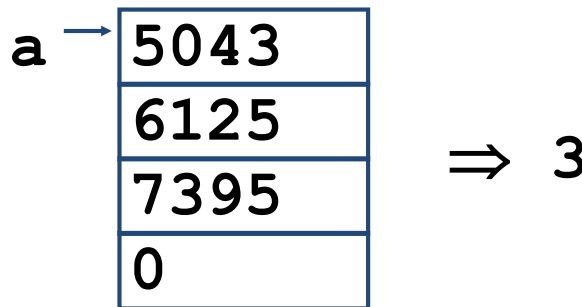
Y86 Code Generation Example

- **First Try**
  - Write typical array code

```
/* Find number of elements in
   null-terminated list */
int len1(int a[])
{
  int len;
  for (len = 0; a[len];
len++)
      ;
  return len;
}
```

- Compile with `gcc34 -O1 -S`

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

```
L5:
   incl  %eax
   cmpl  $0, (%edx,%eax,4)
   jne L5
```

Y86 Code Generation Example #2

■**Second Try**

- Write with pointer code

```
/* Find number of elements in
   null-terminated list */
int len2(int a[])
{
  int len = 0;
  while (*a++)
      len++;
  return len;
}
```

■**Result**

- Don't need to do indexed addressing

```
.L11:
   incl        %ecx
   movl        (%edx), %eax
   addl        $4, %edx
   testl       %eax, %eax
   jne .L11
```

- Compile with `gcc34 –O1 -S`

Y86 Code Generation Example #3

■**IA32 Code**

■ Setup

```
len2:
  pushl %ebp
  movl %esp, %ebp




  movl 8(%ebp), %edx
  movl $0, %ecx
  movl (%edx), %eax
  addl  $4, %edx
  testl %eax, %eax
  je  .L13
```

■**Y86 Code**

■ Setup

```
len2:
  pushl %ebp        # Save %ebp
  rrmovl %esp, %ebp # New FP
  pushl %esi        # Save
  irmovl $4, %esi   # Constant 4
  pushl %edi        # Save
  irmovl $1, %edi   # Constant 1
  mrmovl 8(%ebp), %edx # Get a
  irmovl $0, %ecx      # len = 0
  mrmovl (%edx), %eax  # Get *a
  addl %esi, %edx      # a++
  andl %eax, %eax      # Test *a
  je Done   # If zero, goto Done
```

- **Need constants 1 & 4**
- **Store in callee-save registers**

- **Use `andl` to test register**

Y86 Code Generation Example #4

■**IA32 Code**

■ Loop

```
.L11:
  incl %ecx
  movl (%edx), %eax
  addl $4, %edx
  testl %eax, %eax
  jne .L11
```

■**Y86 Code**

■ Loop

```
Loop:
  addl %edi, %ecx      # len++
  mrmovl (%edx), %eax  # Get *a
  addl %esi, %edx      # a++
  andl %eax, %eax      # Test *a
  jne Loop      # If !0, goto Loop
```

Y86 Code Generation Example #5

- **IA32 Code**
  - Finish

```
.L13:
  movl %ecx, %eax



  leave


  ret
```

- **Y86 Code**
  - Finish

```
Done:
    rrmovl %ecx, %eax# return len
    popl %edi      # Restore %edi
    popl %esi      # Restore %esi
    rrmovl %ebp, %esp # Restore SP
    popl %ebp         # Restore FP
    ret
```

Y86 Sample Program Structure #1

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

    .align 4            # Program data
array:
    . . .


Main:                   # Main function

    . . .
    call len2
    . . .


len2:                   # 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 Program Structure #2

```
init:
    irmovl Stack, %esp   # Set up SP
    irmovl Stack, %ebp   # Set up FP
    call Main            # Execute main
    halt                 # Terminate

# Array of 4 elements + terminating
0
    .align 4
array:
    .long 0x000d
    .long 0x00c0
    .long 0x0b00
    .long 0xa000
    .long 0
```

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

Y86 Program Structure #3

```
Main:
    pushl %ebp
    rrmovl %esp,%ebp
    irmovl array,%edx
    pushl %edx          # Push array
    call len2           # Call
len2(array)
    rrmovl %ebp,%esp
    popl %ebp
    ret
```

- **Set up call to len2**
    - Follow IA32 procedure conventions
    - Push array address as argument

Assembling Y86 Program

```
unix> yas len.ys
```

- **Generates "object code" file `len.yo`**
  - Actually looks like disassembler output

```
0x000:                   |     .pos 0
0x000: 30f400010000 | init:   irmovl Stack, %esp  # Set up stack pointer
0x006: 30f500010000 |    irmovl Stack, %ebp      # Set up base pointer
0x00c: 8028000000   |    call Main               # Execute main program
0x011: 00            |    halt                     # Terminate program
                      |
                      | # Array of 4 elements + terminating 0
0x014:                |    .align 4
0x014:                | array:
0x014: 0d000000      |    .long 0x000d
0x018: c0000000      |    .long 0x00c0
0x01c: 000b0000      |    .long 0x0b00
0x020: 00a00000      |    .long 0xa000
0x024: 00000000      |    .long 0
```

Simulating Y86 Program

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

■ **Instruction set simulator**

▪ Computes effect of each instruction on processor state

▪ Prints changes in state from original

```
Stopped in 50 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:                    0x00000000    0x00000004
%ecx:                    0x00000000    0x00000004
%edx:                    0x00000000    0x00000028
%esp:                    0x00000000    0x00000100
%ebp:                    0x00000000    0x00000100

Changes to memory:
0x00ec:                  0x00000000    0x000000f8
0x00f0:                  0x00000000    0x00000039
0x00f4:                  0x00000000    0x00000014
0x00f8:                  0x00000000    0x00000100
0x00fc:                  0x00000000    0x00000011
```

CISC Instruction Sets

- Complex Instruction Set Computer
- Dominant style through mid-80's

Stack-oriented instruction set

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

Arithmetic instructions can access memory

- `addl %eax, 12(%ebx,%ecx,4)`
  - requires memory read and write
  - Complex address calculation

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)

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 `mrmovl` and `rmmovl`

MIPS Registers

$0	$0	Constant 0
$1	$at	Reserved Temp.
$2	$v0	Return Values
$3	$v1	
$4	$a0	
$5	$a1	Procedure arguments
$6	$a2	
$7	$a3	
$8	$t0	
$9	$t1	
$10	$t2	Caller Save Temporaries:
$11	$t3	May be overwritten by
$12	$t4	called procedures
$13	$t5	
$14	$t6	
$15	$t7	

$16	$s0	
$17	$s1	
$18	$s2	Callee Save
$19	$s3	Temporaries:
$20	$s4	May not be
$21	$s5	overwritten by
$22	$s6	called procedures
$23	$s7	
$24	$t8	Caller Save Temp
$25	$t9	
$26	$k0	Reserved for
$27	$k1	Operating Sys
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$s8	Callee Save Temp
$31	$ra	Return Address

MIPS Instruction Examples

R-R

Op	Ra	Rb	Rd	00000	Fn

```
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 << 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
```

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
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power
    - Most cell phones use ARM processor

Summary

- **Y86 Instruction Set Architecture**
  - Similar state and instructions as IA32
  - Simpler encodings
  - Somewhere between CISC and RISC
- **How Important is ISA Design?**
  - Less now than before for desktops
    - With enough hardware, can make almost anything go fast
  - But very important for mobile devices
  - Intel has evolved from IA32 to x86-64
    - Uses 64-bit words (including addresses)
    - Adopted some features found in RISC
      - More registers (16)
      - Less reliance on stack
      - RISC micro-ops

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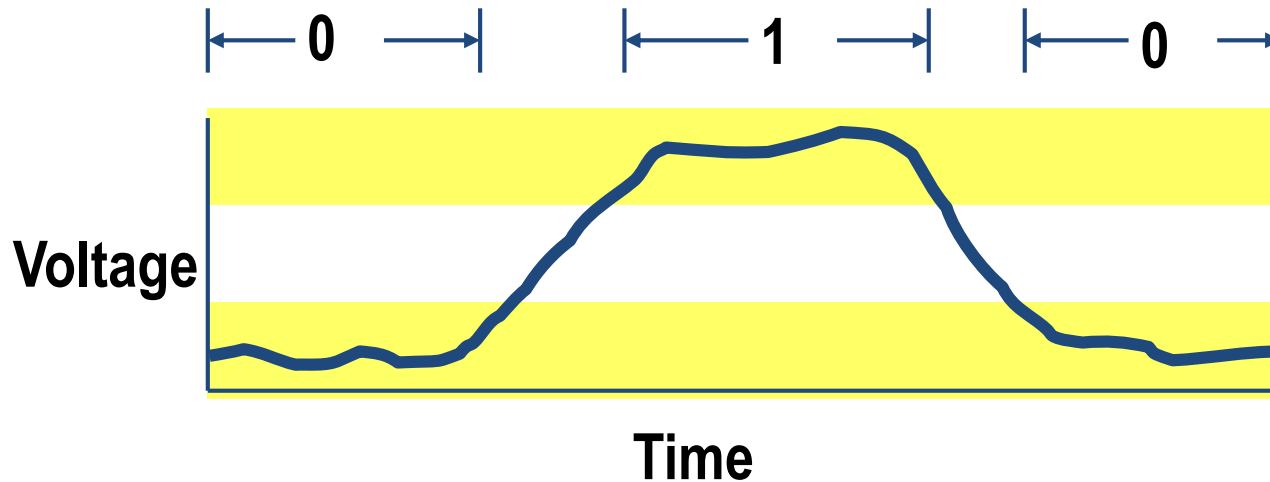
- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

Computing with Logic Gates

And

Or

Not

a

b

out

out = a && b

a

b

out

out = a || b

a
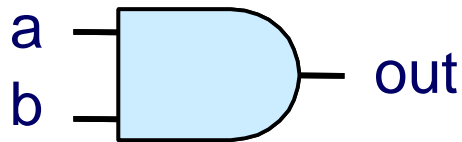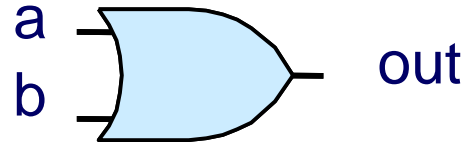
out
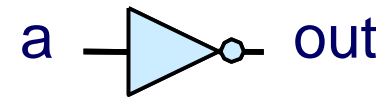
out = !a

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

Rising Delay

Falling Delay

a && b

b

Voltage

a

Time
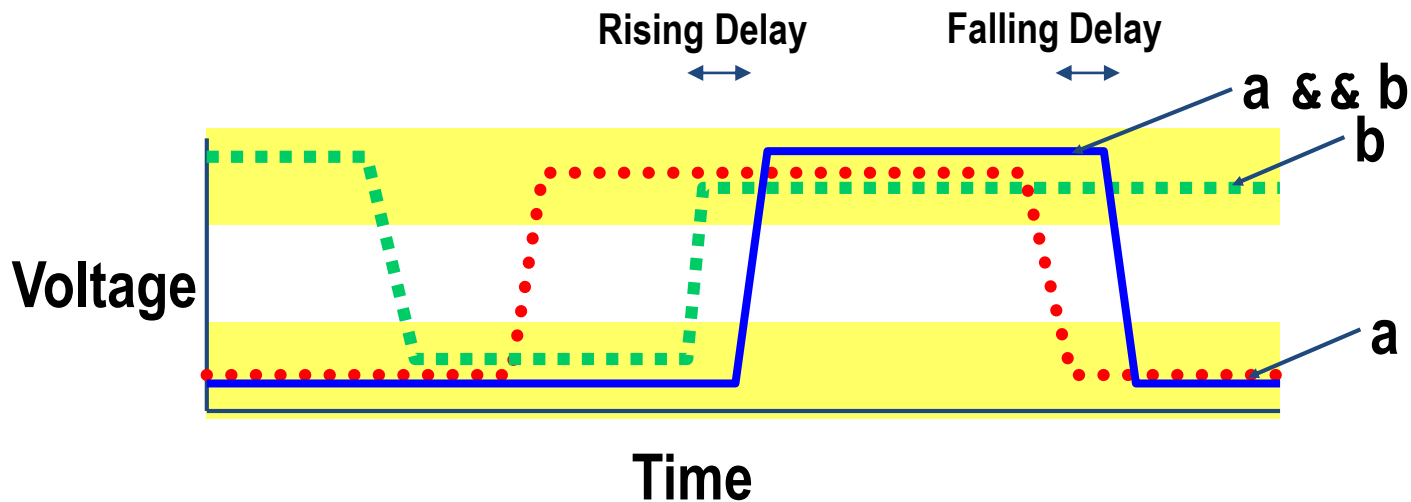
Combinational Circuits

Acyclic Network

Primary Inputs
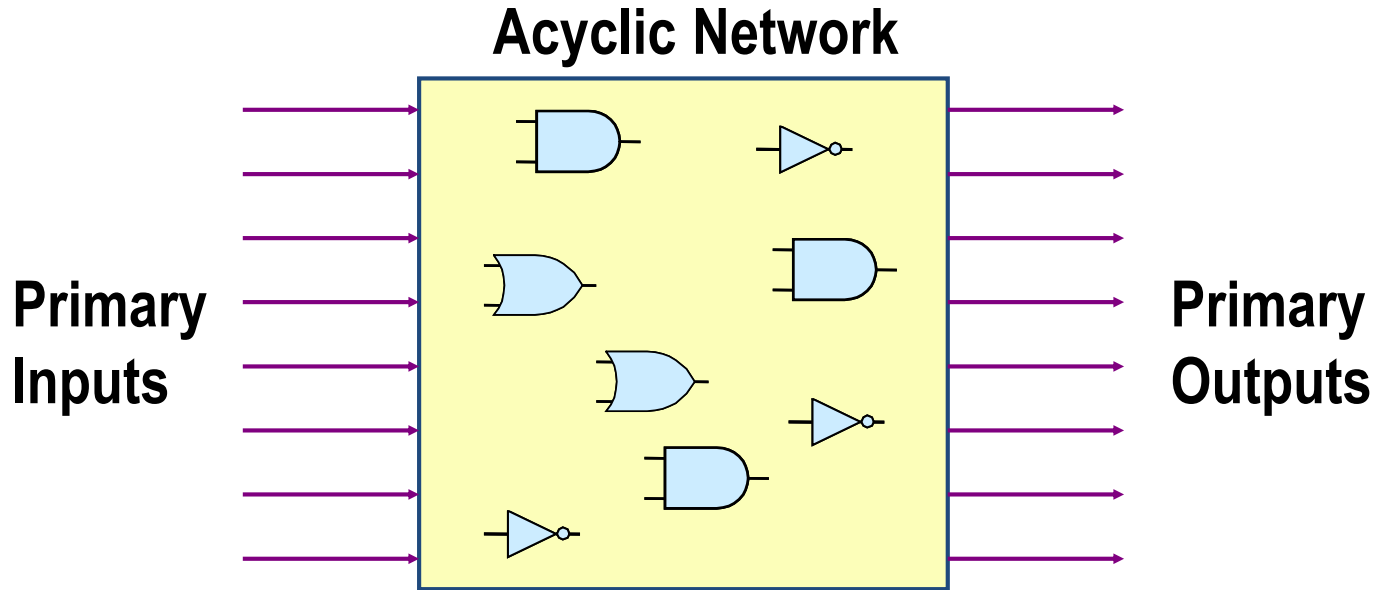
Primary Outputs

■ **Acyclic Network of Logic Gates**

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

Bit Equality
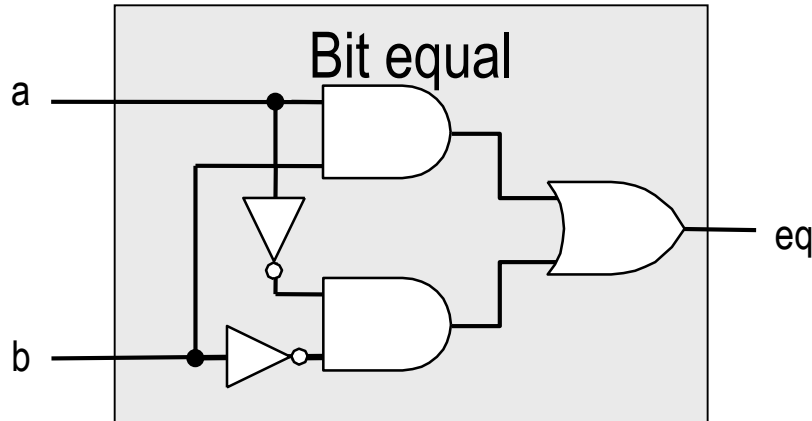
HCL Expression

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

- 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

B —— $=$ —— Eq

A ——
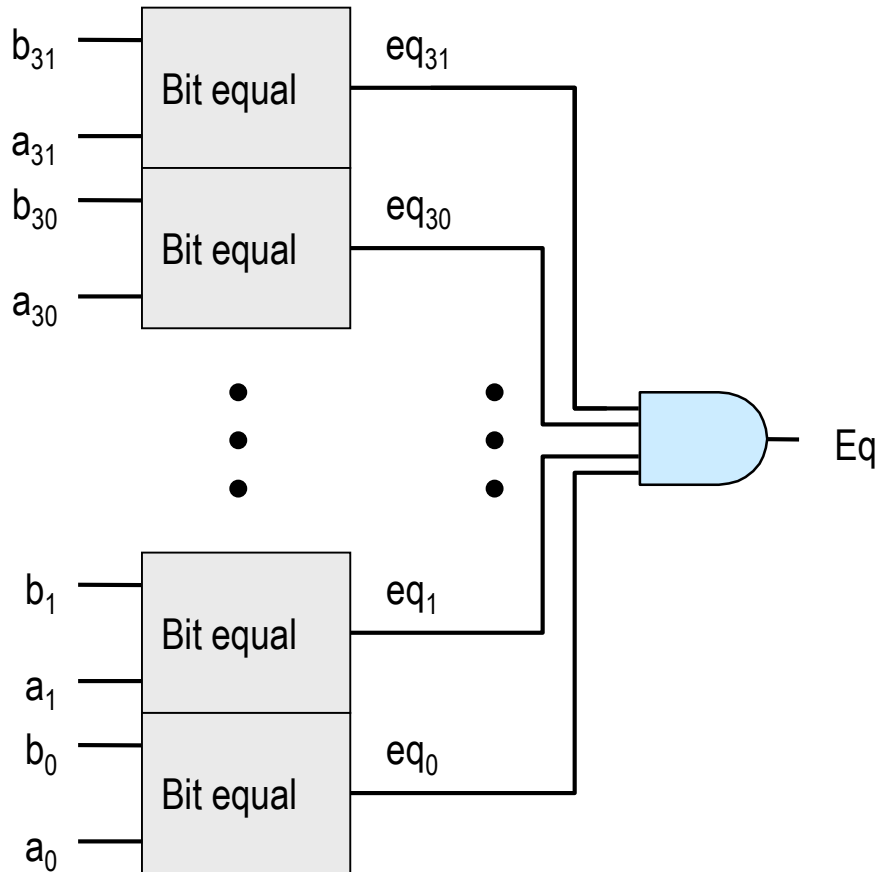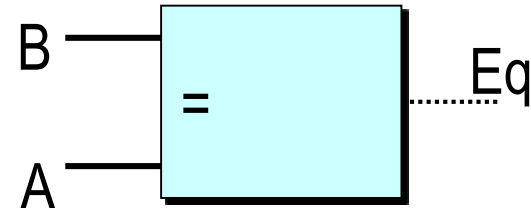
HCL Representation

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

- **32-bit word size**

- **HCL representation**
  - Equality operation
  - Generates Boolean value

Bit-Level Multiplexor
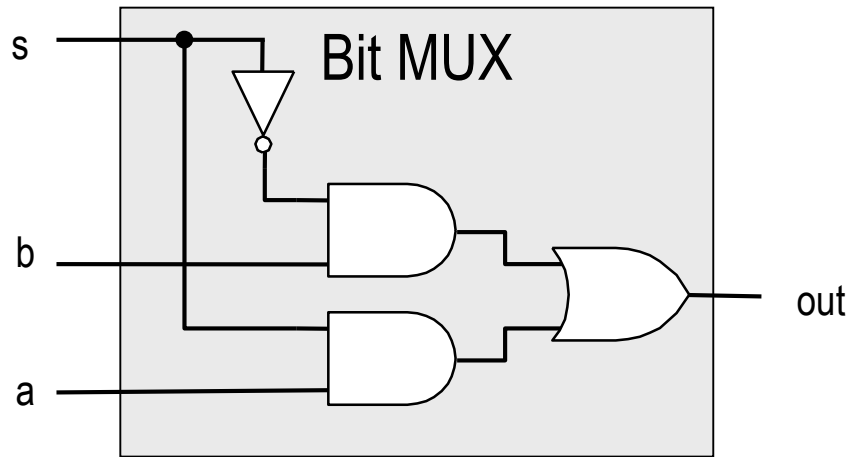
HCL Expression

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

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

46

Word Multiplexor

s

b_{31}

a_{31}

b_{30}

a_{30}

b_0

a_0

out_{31}

out_{30}

out_0

Word-Level Representation

s

B

A

MUX

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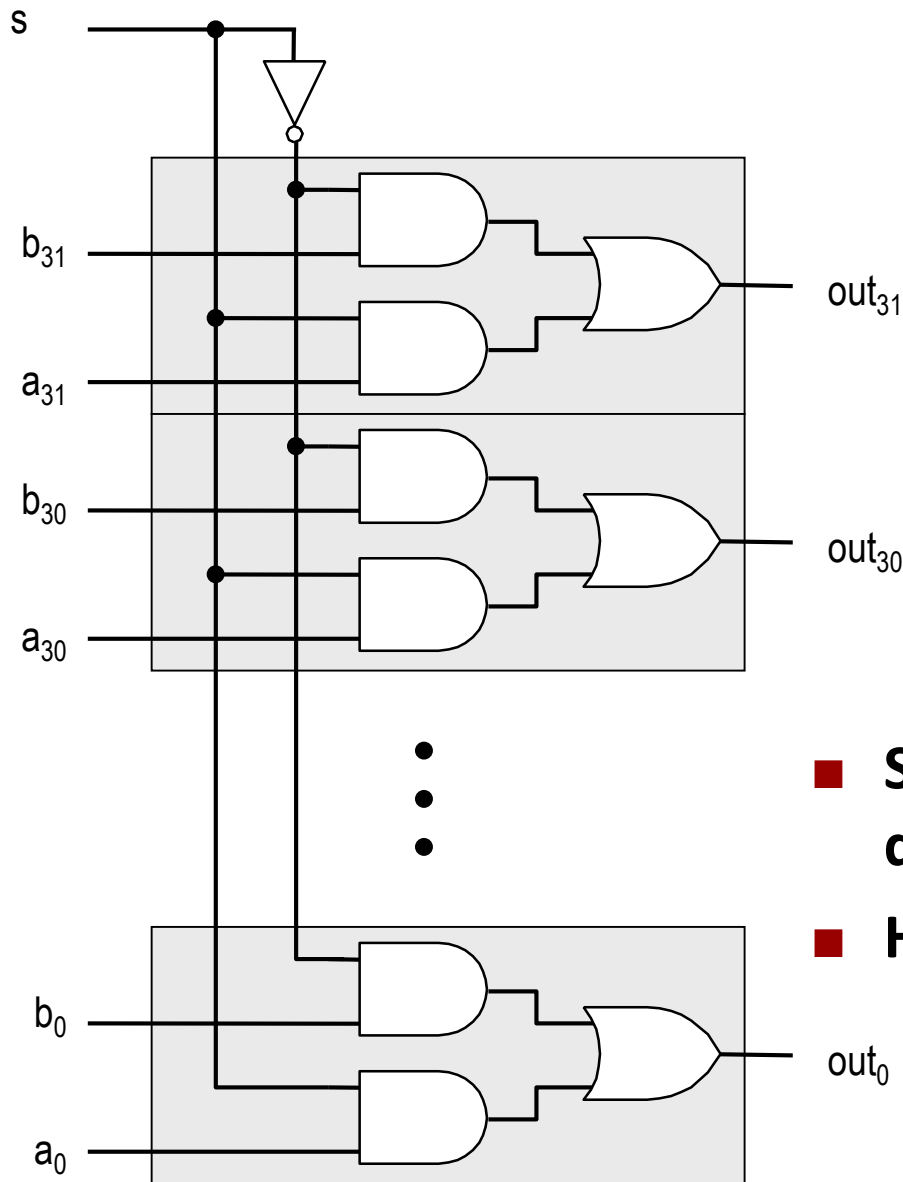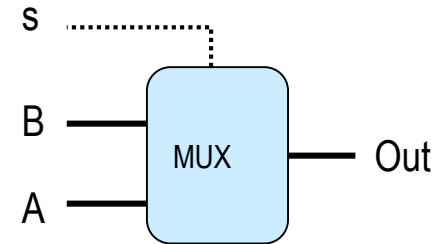
HCL Representation
```
int Out = [
    s : A;
    1 : B;
];
```

- **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

C
B
A
MIN3
Min3
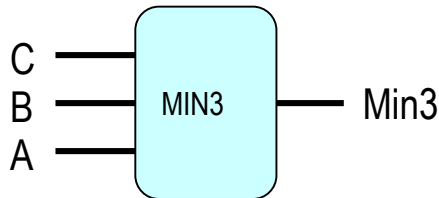
```
int Min3 = [
  A<B && A<C :A;
  B<A && B<C :B;
  1          :C;
];
```

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

4-Way Multiplexor

s1
s0

D0
D1
MUX4
Out4
D2
D3
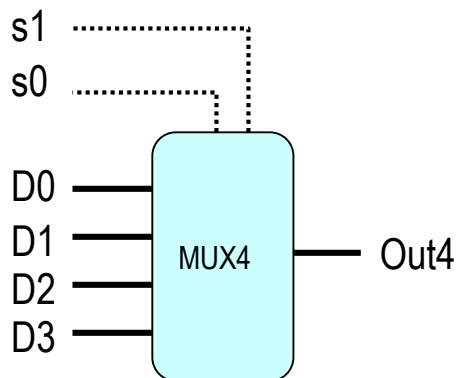
```
int Out4 = [
  !s1&&!s0: D0;
  !s1     : D1;
  !s0     : D2;
  1       : D3;
];
```

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

Arithmetic Logic Unit
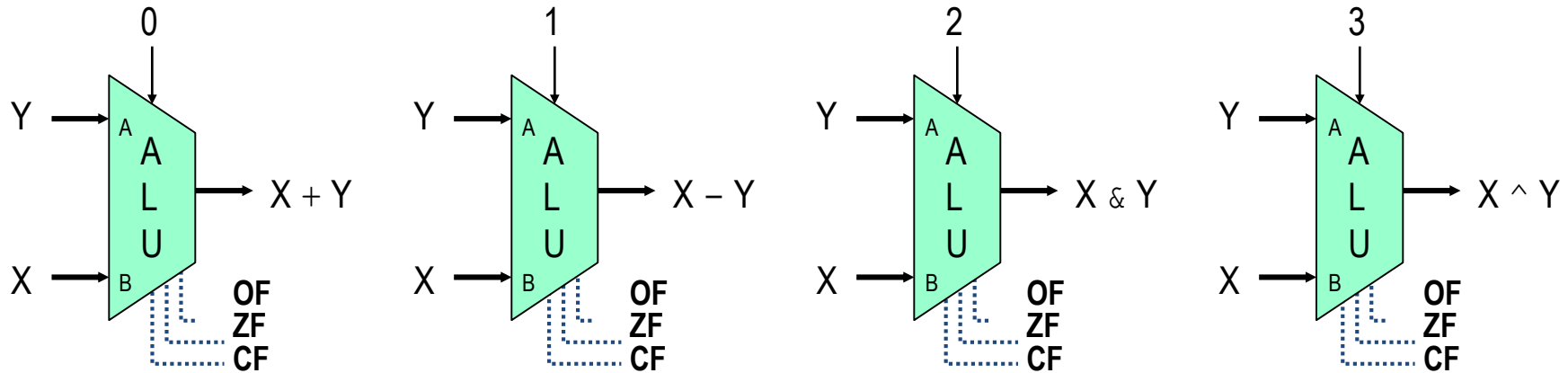
- **Combinational logic**
  - Continuously responding to inputs

- **Control signal selects function computed**
  - Corresponding to 4 arithmetic/logical operations in Y86

- **Also computes values for condition codes**

Storing 1 Bit

Bistable Element

q
Q+

!q
Q−

q = 0 or 1

V_2

V_{in}
V_1
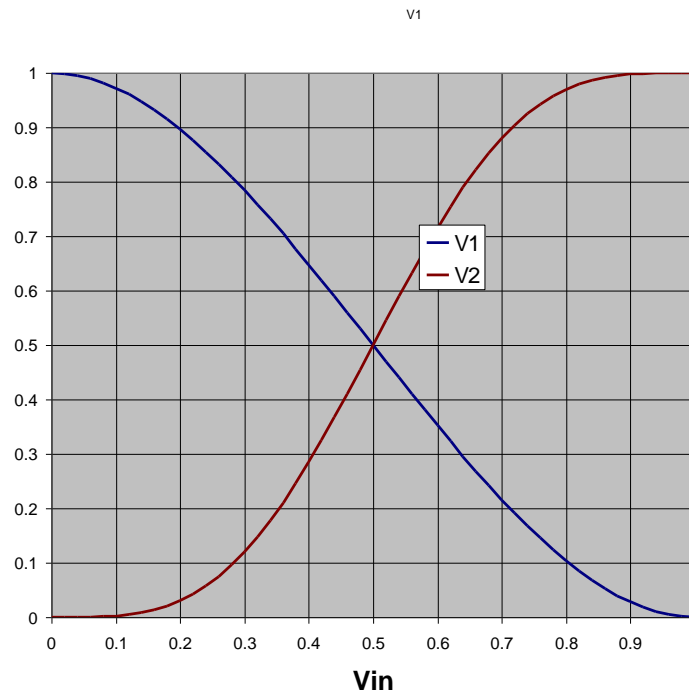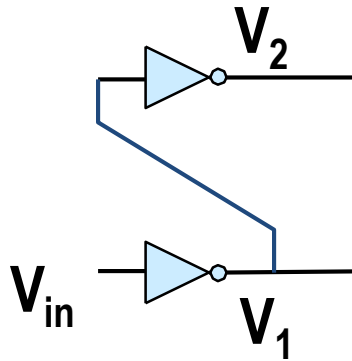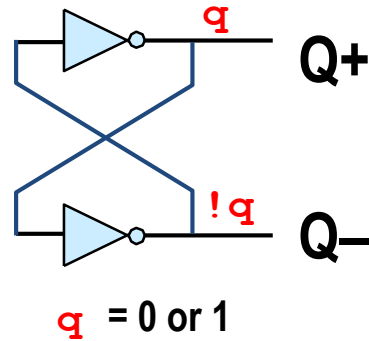
Storing 1 Bit (cont.)

Bistable Element

q

Q+

!q

Q−

q = 0 or 1

Stable 1

$V_{in} = V_2$

V_2

V_{in}

V_1

Metastable

Stable 0

Vin

51
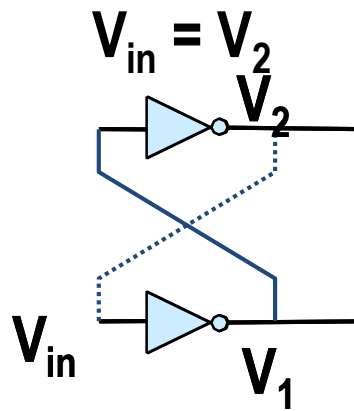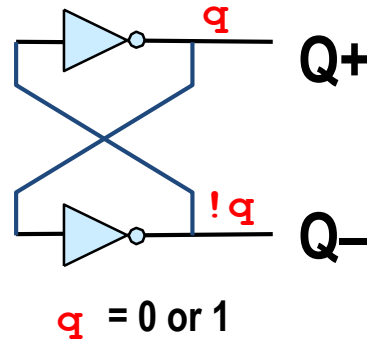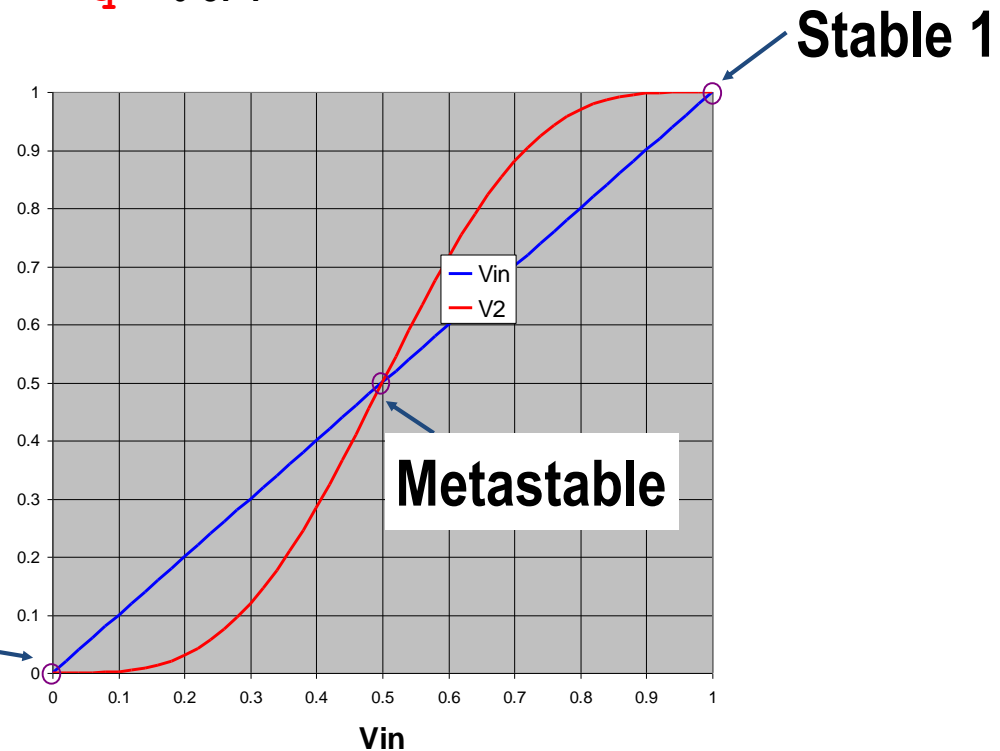
Storing and Accessing 1 Bit

Bistable Element

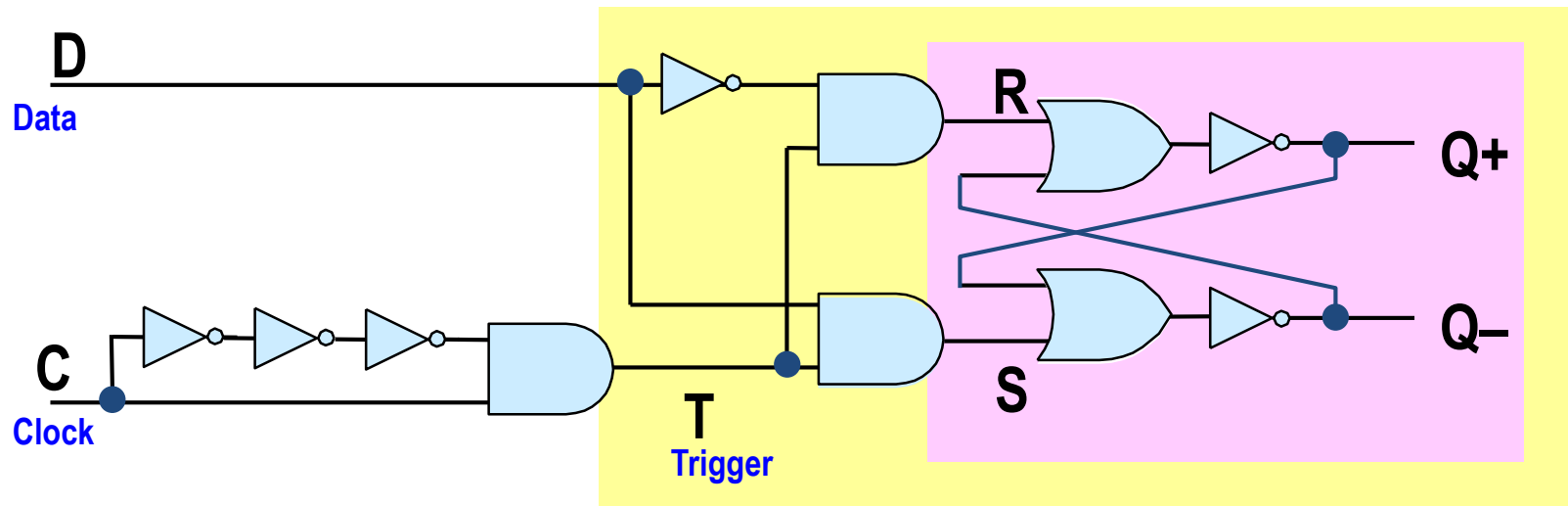q = 0 or 1
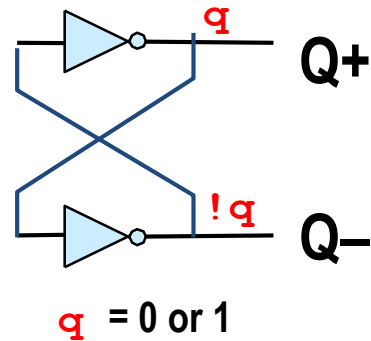
R-S Latch
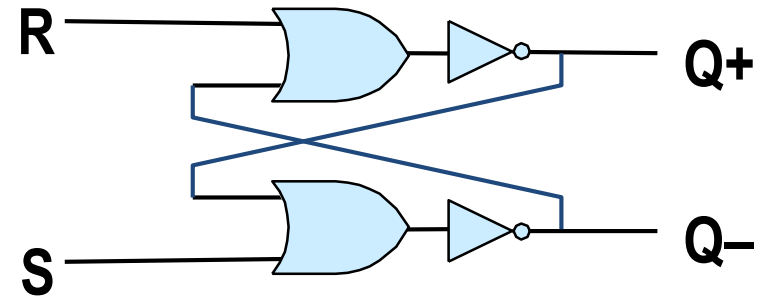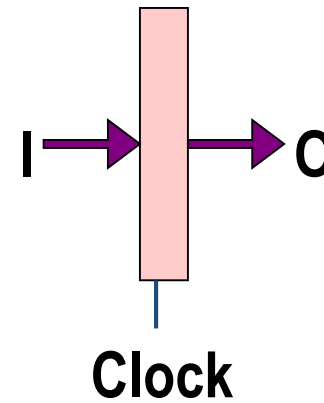

Resetting
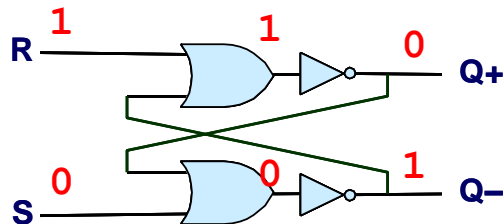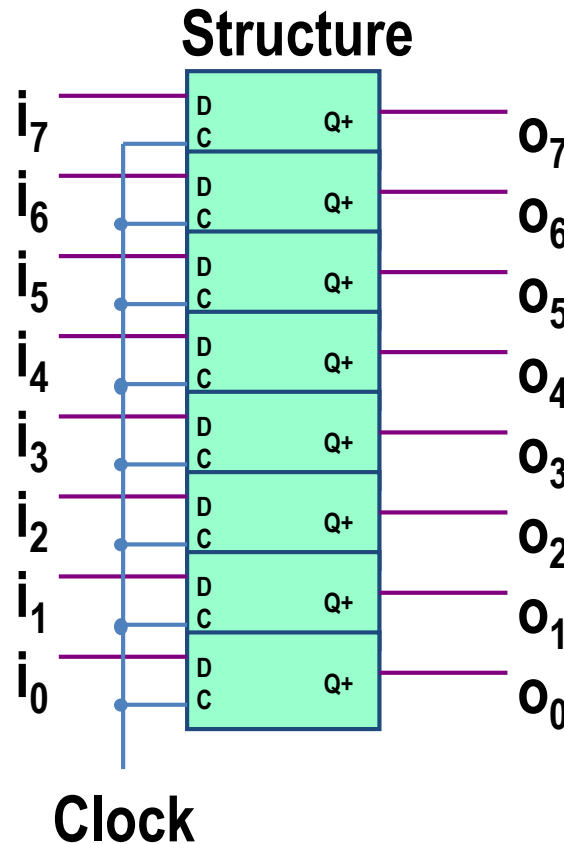

Setting
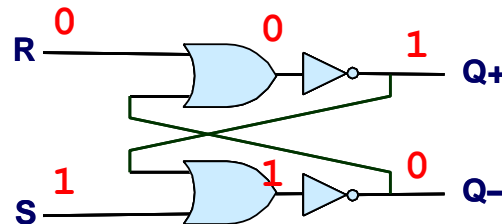

Storing
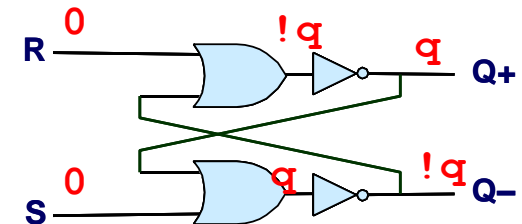
Edge-Triggered Latch
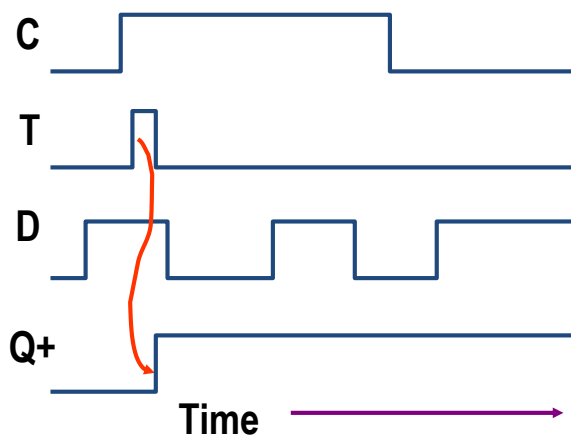
- **Only in latching mode for brief period**
  - Rising clock edge
- **Value latched depends on data as clock rises**
- **Output remains stable at all other times**
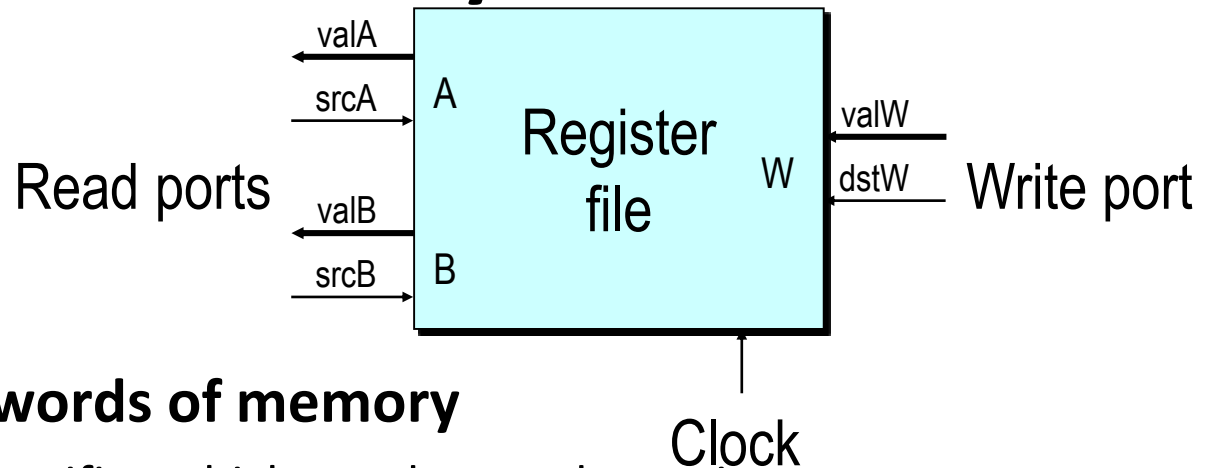
Registers

Structure

- **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

State = x

Input = y   Output = x

x

⇨

Rising
clock

⇨

State = y

Output = y

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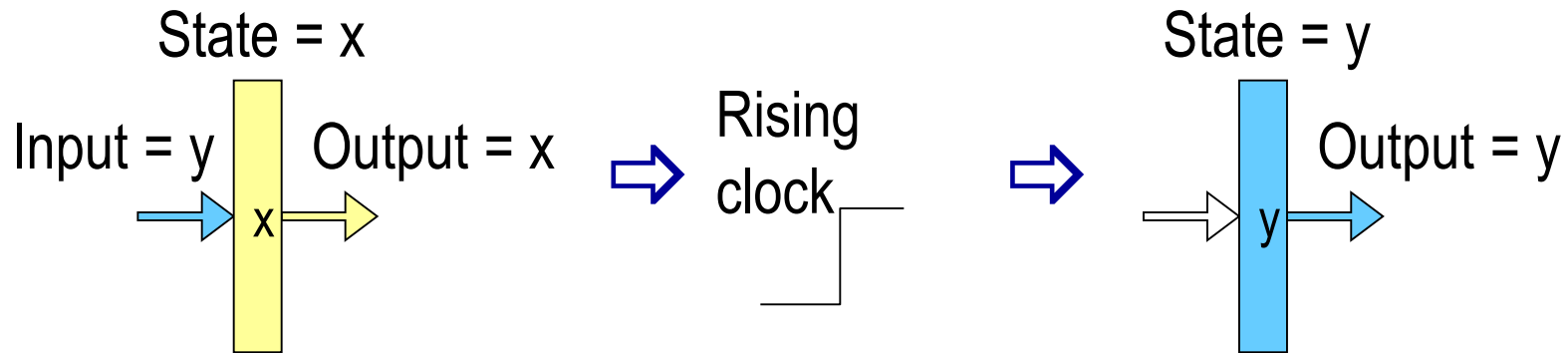
- **Stores data bits**
- **For most of time acts as barrier between input and output**
- **As clock rises, loads input**

Random-Access Memory

valA

srcA   A

Read ports   Register file   W   valW

dstW   Write port

valB

srcB   B

Clock

- **Stores multiple words of memory**
  - Address input specifies which word to read or write
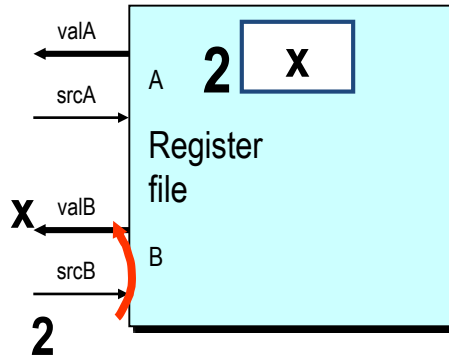
- **Register file**
  - Holds values of program registers
  - `%eax`, `%esp`, etc.
  - Register identifier serves as address
    - ID 15 (0xF) implies no read or write performed

- **Multiple Ports**
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output
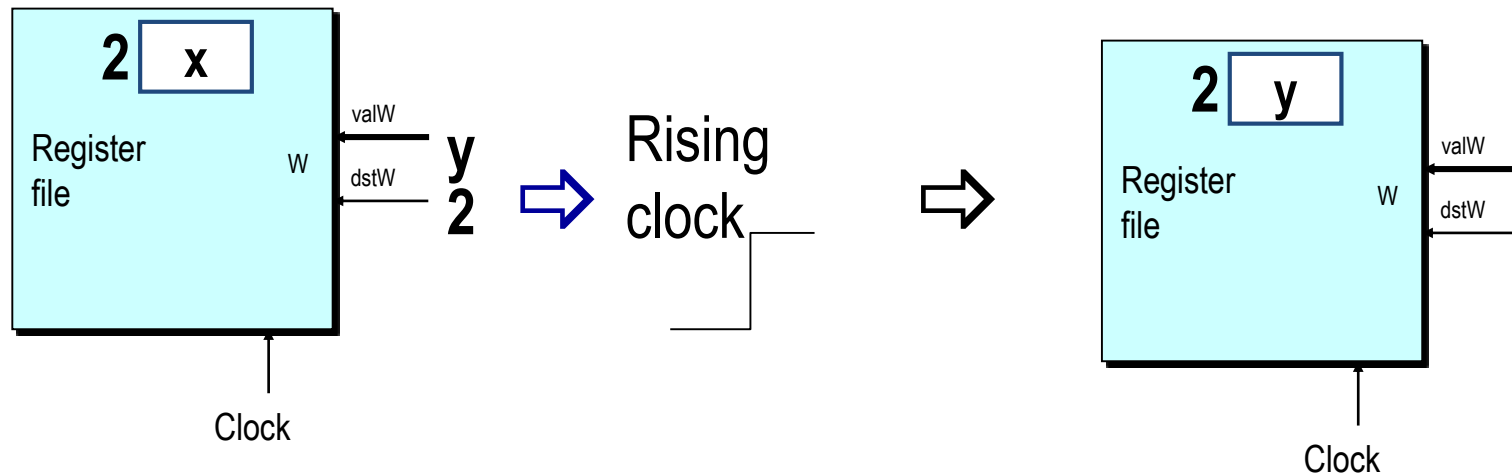
Register File Timing

- **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, 32-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