

A Tale of Two For Loops:

Behold the C code for two functionally equivalent for loops:

```
LOOP 1
for(i=0;i<SIZE;i++) {
    arr2[i]=arr1[i];
}
```

```
LOOP 2
for(i=0;i<SIZE;i+=5) {
    arr2[i]=arr1[i];
    arr2[i+1]=arr1[i+1];
    arr2[i+2]=arr1[i+2];
    arr2[i+3]=arr1[i+3];
    arr2[i+4]=arr1[i+4];
}
```

Question: Which loop is faster? Why?

Now, let's look at the corresponding IA-32 code for the body of each loop with no optimizations applied:

LOOP 1:

```
movl    -12(%ebp), %eax
movl    -12(%ebp), %edx
movl    -40028(%ebp,%edx,4), %edx
movl    %edx, -80028(%ebp,%eax,4)
incl    -12(%ebp)
```

LOOP 2:

```
movl    -12(%ebp), %edx
movl    -12(%ebp), %eax
movl    -40028(%ebp,%eax,4), %eax
movl    %eax, -80028(%ebp,%edx,4)
movl    -12(%ebp), %edx
incl    %edx
movl    -12(%ebp), %eax
incl    %eax
movl    -40028(%ebp,%eax,4), %eax
movl    %eax, -80028(%ebp,%edx,4)
movl    -12(%ebp), %edx
addl    $2, %edx
movl    -12(%ebp), %eax
addl    $2, %eax
movl    -40028(%ebp,%eax,4), %eax
movl    %eax, -80028(%ebp,%edx,4)
movl    -12(%ebp), %edx
addl    $3, %edx
movl    -12(%ebp), %eax
addl    $3, %eax
movl    -40028(%ebp,%eax,4), %eax
movl    %eax, -80028(%ebp,%edx,4)
```

```

movl    -12(%ebp), %edx
addl    $4, %edx
movl    -12(%ebp), %eax
addl    $4, %eax
movl    -40028(%ebp,%eax,4), %eax
movl    %eax, -80028(%ebp,%edx,4)
addl    $5, -12(%ebp)

```

Question: I ask again, which loop is faster? Why?

Depending on the system, loop 2 runs over twice as fast as loop 1. Why? It depends. Let's take a look for SIZE=10000:

	mov instructions	Arithmetic	Compares+Cond. Jumps	Iterations
#1	4	1	1	10000
#2	20	9	1	2000

	Eff. mov instructions	Eff Arith.	Eff. Compares	Total
#1	40000	10000	10000	60000
#2	40000	18000	2000	60000

As you will see, the answer is more subtle than LOOP 2 is accomplishing more per loop. As we turn our gaze towards the design of the CPU, we will see among other things that the conditional jump operations can be comparatively expensive depending on how it is implemented inside the CPU. Until now, we have considered each assembly instruction as an atomic unit, however, each instruction has multiple stages. We call the operation cycle of the instructions the **instruction cycle**

Instruction Cycle

We will look at each cycle in more detail later, but each instruction that is processed follows these stages of operation:

- **Fetch:** Using the program counter, the instruction is fetched from memory and loaded into *instruction register*
- **Decode:** The operands are interpreted and read from either the register file or from memory.
- **Execute:** The ALU performs the operation designated by the instruction. Condition registers are set
- **Memory:** Reading and writing data from memory.
- **Write Back:** Registers are updated.
- **PC update:** Program counter is incremented

RISC and CISC

There are two general approaches to designing an instruction set architecture, *Reduced Instruction Set*




Computing also known as **RISC** (pronounced "risk") or *Complex Instruction Set Computing* also known as **CISC** (pronounced "sisk"). Historically, CISC architectures emerged first and RISC developed in response. Today, many architectures use elements of both approaches. We look at some of the properties of each in the following table:

	CISC	RISC
Instruction encoding:	Variable length encodings.	Fixed-length encodings. Generally, all instructions are encoded as 4 bytes
Addressing format	Multiple, potentially complex formats.	Simple format, often only base address+offset
ALU	ALU operations use both registers & memory locations	ALU operations use only registers, must explicitly load memory locations into a register and store results from registers back to memory
ISA size	Typically very large. Manuals may be > 1K in length!	Fewer instructions, usually less than 100
Execute time	A single instruction may take multiple clock cycles to execute	Instructions take a single clock cycle to execute

... and much, much more. (see pages 324-343 in your textbook).

Logic Gates

Regardless of the design philosophy behind an architecture, the fundamental unit of the digital circuits that comprise the CPU is the logic gate. **Logic gates** are physical circuits that perform a logical operation on one or more inputs and produce a single output. The three basic logic gates are the AND, OR and NOT gates:

Type	Symbol	HCL expression	Truth Table															
AND		A && B	<table><tr><th>A</th><th>B</th><th>A && B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	A && B	0	0	0	0	1	0	1	0	0	1	1	1
A	B	A && B																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		A B	<table><tr><th>A</th><th>B</th><th>A B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	A B	0	0	0	0	1	1	1	0	1	1	1	1
A	B	A B																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		!A	<table><tr><th>A</th><th>!A</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	!A	0	1	1	0									
A	!A																	
0	1																	
1	0																	

But there is more...

By assembling more than one logic gate into a network, we can construct **combinational circuits**. A combinational circuit can be built from any combination and sequence of logic gates given we observe the following two constraints:

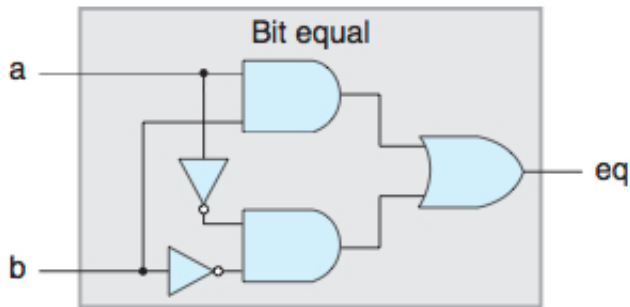
1. The outputs of two or more logic gates cannot be connected together.
2. The network must be *acyclic*, meaning that no path through the network may form a loop.

We use **Hardware Control Language (HCL)** expressions to describe logic gates and combinational circuits.

Example: What is the HCL for testing bit-level equality? Draw the combinational circuit.

```
(a && b) || (!a && ! b)
```

The combinational circuit would look like this:



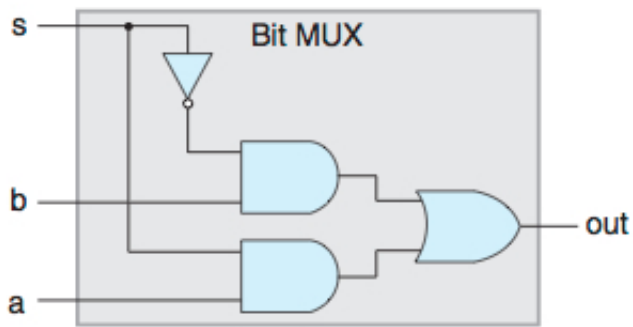
Example: Write an HCL expression for bit XOR.

```
(!a && b) || (a && !b)
```

Multiplexors

Multiplexors select a value among a set of different data signals, depending on the value of a control input signal. In a single-bit multiplexor, inputs A and B are selected based on control signal s. When s is 1, the output of the mutliplexor will be A, when s is 0, the output is B. The HCL expression and diagram of the circuit are as follows:

```
(s && a) || (!s && b)
```

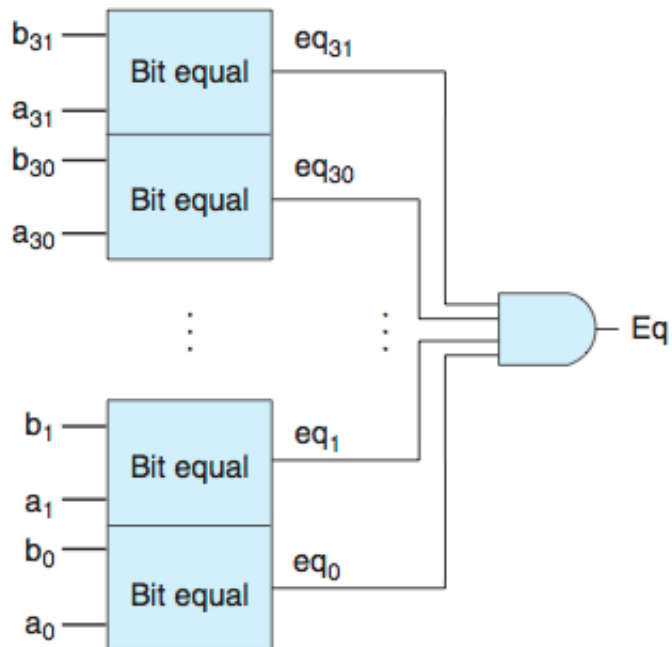


Truth Table:

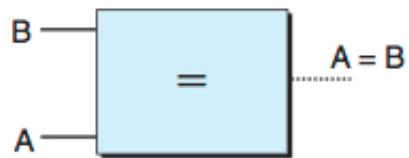
a	b	s	!s	s && a	!s && b	Multiplexor
0	0	0	1	0	0	0 (B)
0	0	1	0	0	0	0 (A)
0	1	0	1	0	1	1 (B)
0	1	1	0	0	0	0 (A)
1	0	0	1	0	0	0 (B)
1	0	1	0	1	0	1 (A)
1	1	0	1	0	1	1 (B)
1	1	1	0	1	0	1 (A)

Word-Level Combinational Circuits

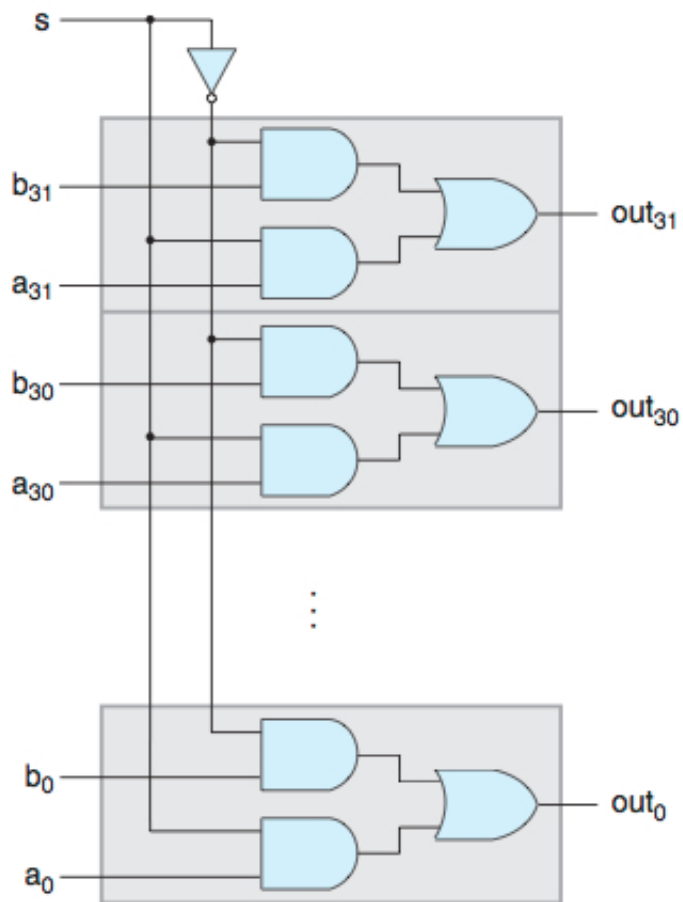
We can build word-level combinational circuits. Word-level equality:



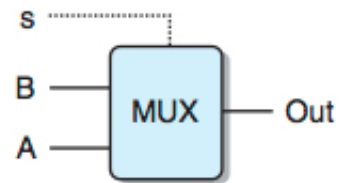
(a) Bit-level implementation



(b) Word-level abstraction



(a) Bit-level implementation



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

(b) Word-level abstraction