RUTGERS

THE STATE UNIVERSITY OF NEW JERSEY

CS 516 Compilers and Programming Languages II

Reversible Computing-2
Quantum Computing-1

Annoucements

Tuesday, April 28:

→ Uncertain<T>, Hari
→ Sarana, Xiao
→ JouleGuard, Fei

Friday, May 1

→ DNA programming, Thomas
→ Neuromorphic computing, Jonathan
→ Quantum computing status, Jiapeng

Papers (for April 28) are posted under sakai/resources;
Overviews (for May 1) need independent research

Once you read your paper or did some research, send me and email and we will talk.

Why look at reversible computing?

The **second law of thermodynamics:**
Total entropy of an isolated system can never decrease over time, and is constant if and only if all processes are reversible.

Losing information from a digital system by erasing/overwriting bits **necessarily** implies ejecting that information into the system's environment.

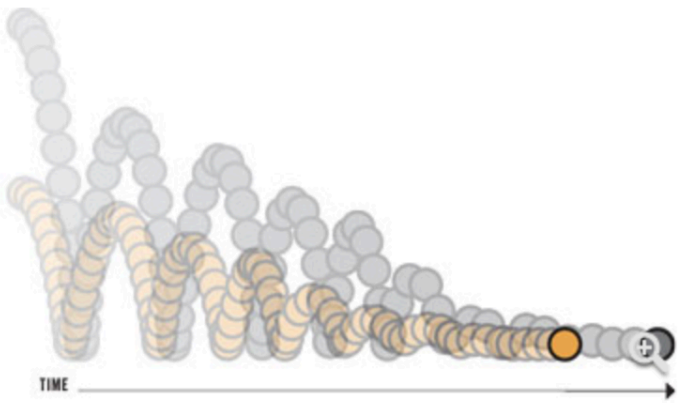
Once "absorbed" into the environment, information that was previously known (correlated) becomes entropy, i.e., unknown / uncorrelated information.

Why look at reversible computing?

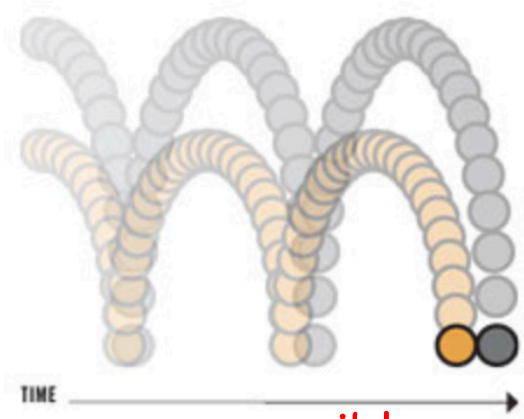
Irreversible systems lose information which is directly related to dissipating energy into the environment (increasing entropy)

Example: irreversible: drop a ball - it will end up on the ground, but from what height was it thrown?

reversible: ideal situation without friction – will return to the height it was dropped from



irreversible



reversible

information loss == energy loss, i.e.,
deleting information "costs" / dissipates energy

Why look at reversible computing?

How does irreversible computing relate to current computing?

Current computation use CMOS where erasing a bit leads to power dissipation (registers, caches, memory, …). The final answer of the computation typically does not let us determine its input in a deterministic way.

Example: Observation: Your program prints '5'

Question: What was the input for your program?

Current computation is forward computation only. Reversible computing includes forward and backward computation.

Why look at reversible computing?

Theoretical benefits: No information is lost in the system, which means that no energy is dissipated.

Noise to signal ratios: Irreversible systems have a theoretical fixed limit on how much power must be dissipated for a processing a piece of information (e.g, a bit switch). This is called the Landauer's limit: k_B T ln 2 (k_B - Boltzmann constant, T - temperature) and results in 1 electron volt per bit operation energy cost at room temperature (20°C).

Current technology is FAR away from this limit, but the limit is real and holds for ALL computing technologies

Reversible computing does not have such a theoretical limit.

Reversible programming languages

A reversible computing model allows deterministic, time-invertible computation, in which not only the next computation state, but also the previous computation state is determined uniquely by the current state

- ⇒ all computations are forward and backward deterministic
- ⇒ store updates are non-destructive, i.e., do not lose information

(T. Yokoyama, H.B. Axelsen, and R. Glück, 2008)

My interpretation:

Reversible computation gives you a chance to look at the same information in different ways ("information isomorphism"), starting with the input and ending with the output across the entire state sequence ("observation spectrum") .

Fundamental definitions / results

(1) A function f is injective iff $f(x) = f(y) \Rightarrow x = y$

(2) Since reversible languages cannot compute non-injective functions, they are not universal (not Turing complete).

(3) RTM (Reverse Turing Machine): Turing machine (TM) with forward and backward deterministic rules; as reversible languages, RTMs can only compute injective functions. RTMs do not have "garbage" input/output (see below)

(4) Any general TM computation / irreversible computation can be embedded in a RTM if garbage information is allowed (output in addition to the "regular" output – example: return input together with output);

(5) Janus is a reversible, imperative language which is r-Touring-complete, i.e., is as expressive/powerful as a RTM

Basic Operational Semantics

(complete rules for Janus language in posted paper for this lecture)

Store σ: ***variables*** $\rightarrow$ ***values*** , i.e., a mapping from variables to values

Meaning of statement s: $\sigma \vdash_{stmt} \mathbf{s} \Rightarrow \sigma'$

under store σ the execution of ***s*** yields store σ'

All statements are reversible, i.e., for each ***s*** there is a $\mathbf{s}^{-1}$ such that

$$\sigma \vdash_{stmt} \mathbf{s} \Rightarrow \sigma' \text{ and } \sigma' \vdash_{stmt} \mathbf{s}^{-1} \Rightarrow \sigma$$

Example: Procedure calling and un-calling

$$\sigma \vdash_{stmt} \textbf{call } \mathbf{p}(\langle input\_vars \rangle) \Rightarrow \sigma' \text{ and}$$
$$\sigma' \vdash_{stmt} \textbf{uncall } \mathbf{p}(\langle output\_vars \rangle) \Rightarrow \sigma$$

Janus programming language (subset)

- Only integer as basic type and stack as composite type
- Expressions **e** with arithmetic and bitwise logical integer operators
- Limited form of assignment statements

x += e, x -= e, or x ^= e (add, subtract, bitwise exor)

x ⊕= e means x = x ⊕ e for ⊕ in { +, -, ^ }

Inverses: x += e → x -= e

x -= e → x += e

x ^= e → x ^= e

- Swap operation x ⇔ y

  "syntactic sugar" for x ^= y; y ^= x; x ^= y

Janus programming language (subset)

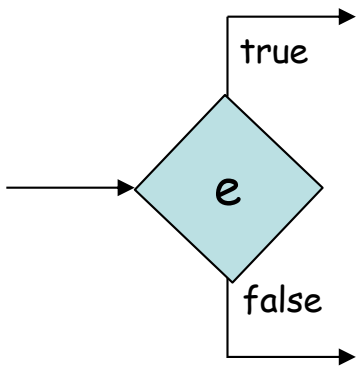
- Assignment statement

- Sequence of statements (possibly nested assign, condition, loop, call)

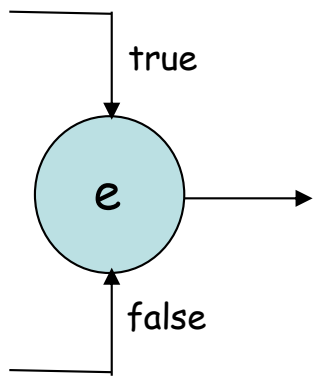
with inverse

Janus programming language (subset)

- Conditional



test(e): dispatch control flow based on value of e

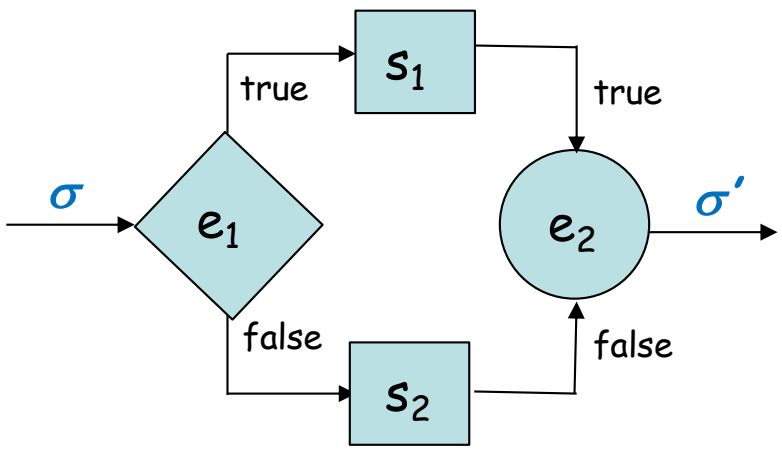


assert(e): value of e must be true if joint point is reached along true-edge, and false if joint point is reached along false edge; Otherwise undefined

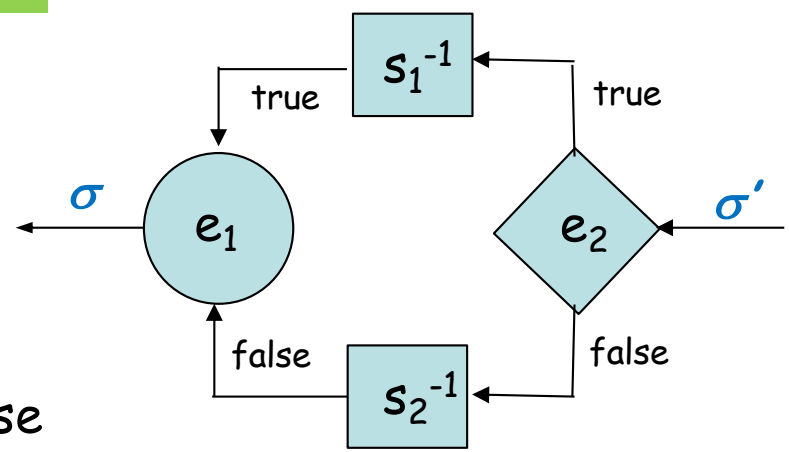
Janus programming language (subset)

- Conditional

```
if e1
  then s1
  else s2
fi e2
```



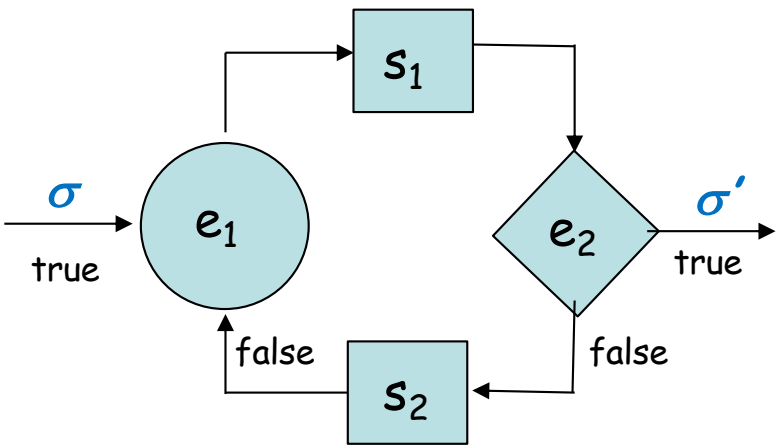
with inverse



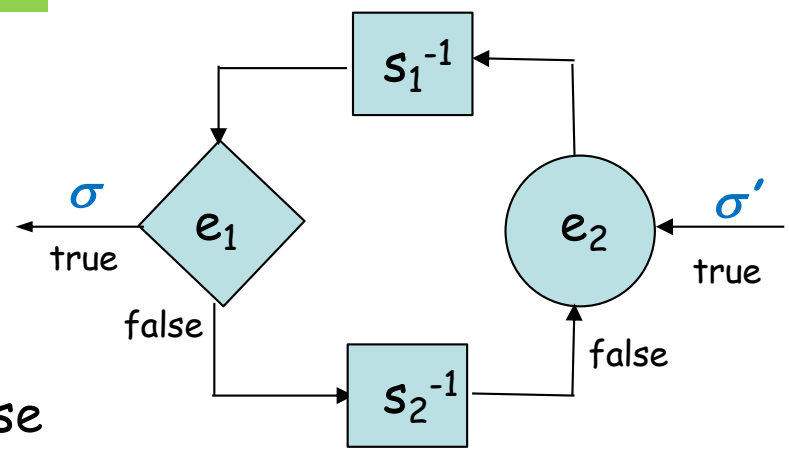
Janus programming language (subset)

- Loop

from e_1
do s_1
loop s_2
until e_2



with inverse



Janus programming language (subset)

Online interpreter:
http://topps.diku.dk/pirc/?id=janusP

- Procedure call
  - → Variables are local (initialized to 0)
  - → Call-by-reference

```
procedure main()
    int x1
    x1 += 100
    call test(x1)

procedure test(int x)
    x+=70
    x-=15
    x+=5
```

```
x1 = 160
```

```
procedure main()
    int x1
    x1 += 160
    uncall test(x1)

procedure test(int x)
    x+=70
    x-=15
    x+=5
```

```
x1 = 100
```

Reversible programming languages

```
procedure fib(int x1, int x2, int n)
    if n = 0 then
        x1 += 1
        x2 += 1
    else
        n -= 1
        call fib(x1, x2, n)
        x1 += x2
        x1 <=> x2
    fi x1 = x2

procedure main()
    int x1
    int x2
    int n
    n += 4
    call fib(x1, x2, n)
```

Run

```
n = 0
x1 = 5
x2 = 8
```

For input "n", return the (n+1)-th and (n+2)-th Fibonacci numbers

n	Fib(n+1) **x1**	Fib(n+2) **x2**
0	1	1
1	1	2
2	2	3
3	3	5
4	5	8
5	8	13
6	13	21

Reversible programming languages

```
procedure fib(int x1, int x2, int n)
    if n = 0 then
        x1 += 1
        x2 += 1
    else
        n -= 1
        call fib(x1, x2, n)
        x1 += x2
        x1 <=> x2
    fi x1 = x2

procedure main()
    int x1
    int x2
    int n
    x1 += 2
    x2 += 3
    uncall fib(x1, x2, n)
```

Run

```
n = 2
x1 = 0
x2 = 0
```

For input "n", return the (n+1)-th and (n+2)-th Fibonacci numbers

n	Fib(n+1) **x1**	Fib(n+2) **x2**
0	1	1
1	1	2
2	2	3
3	3	5
4	5	8
5	8	13
6	13	21

Reversible programming languages

```
4   procedure fib(int x1, int x2, int n)
5       if n = 0 then
6           x1 += 1
7           x2 += 1
8       else
9           n -= 1
10          call fib(x1, x2, n)
11          x1 += x2
12          x1 <=> x2
13      fi x1 = x2
14
15  procedure main()
16      int x1
17      int x2
18      int n
19      x1 += 2
20      x2 += 3
21      uncall fib(x1, x2, n)
```

Run

```
n = 2
x1 = 0
x2 = 0
```

trace

	x1	x2	n
21: uncall fib(x1,x2,n)	2	3	0
13: x1=x2 is false			
12: x1 <=> x2	3	2	0
11: x1 -= x2.	1	2	0
10: uncall fib(x1,x2,n)	1	2	0
-> 13: x1=x2 is false			
12: x1 <=> x2	2	1	0
11: x1 -= x2	1	1	0
10: uncall fib(x1,x2,n)	1	1	0
-> 13: x1=x2 is true			
7: x2 -= 1	1	0	0
6: x1 -= 1	0	0	0
<- 5: n=0 is true			
9: n += 1	0	0	1
<- 5: n=0 is false			
9: n += 1	0	0	2
5: n=0 is false			

Reversible programming languages

Summary

- Janus is an imperative, reversible language; there are other proposals of reversible languages (e.g.: functional)
- Full Janus language contains integer stacks as data types
- Not all programs implement injective functions; however, by adding additional information, any function can be expressed as an injective function
- Limited tools available to develop and debug reversible programs
- Efficient reversible hardware is still in development (e.g.: adiabatic circuits)
- Quantum computing is reversible computing
- Reversible computing is (based on its theoretical foundation) the most energy efficient way of computing

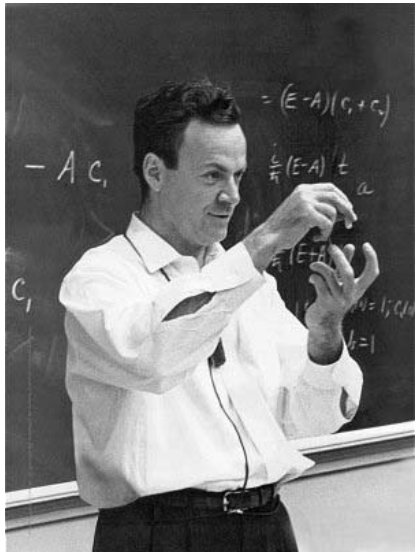
RUTGERS

THE STATE UNIVERSITY
OF NEW JERSEY

CS 516 Compilers and Programming Languages II

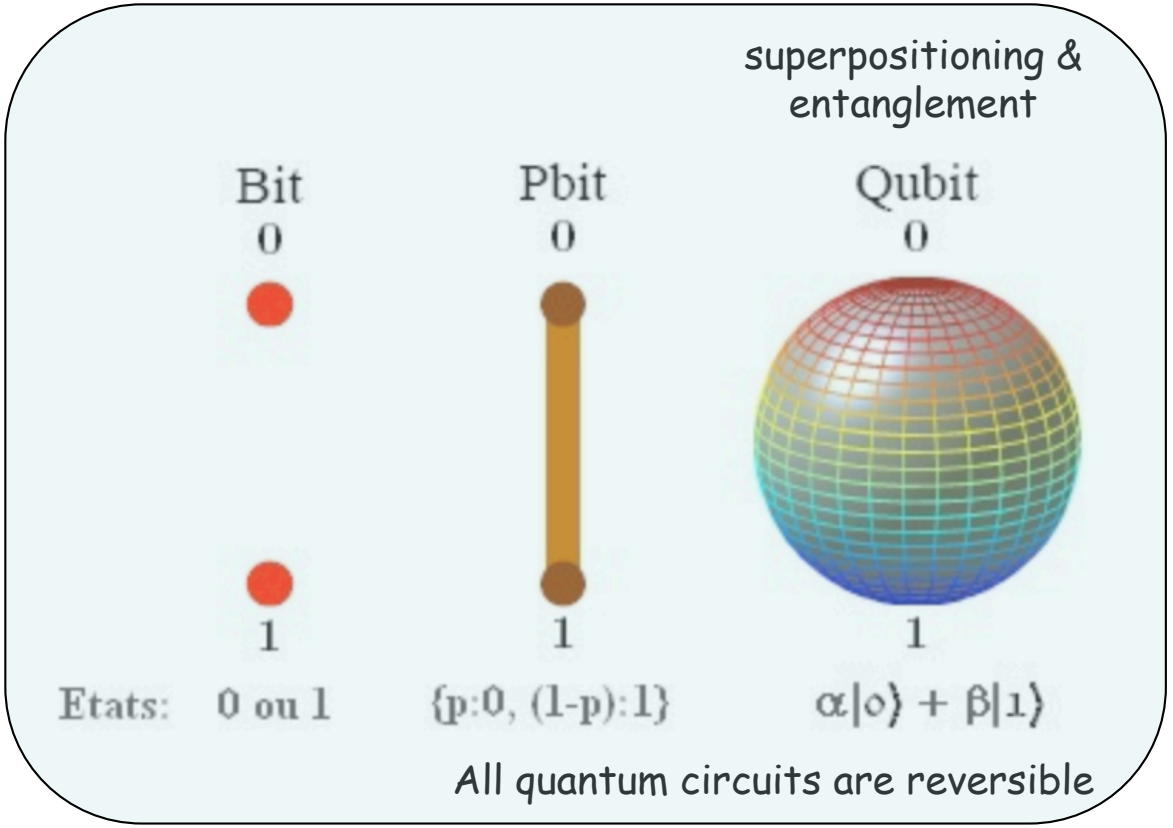
Quantum Computing

Quantum Computing



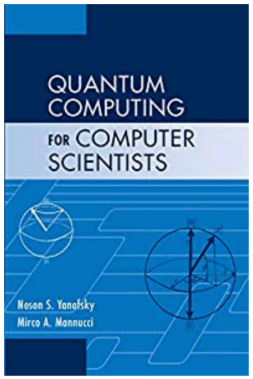
- "I think I can safely say that nobody understands quantum mechanics" – Richard Feynman
- 1982 - Feynman proposed the idea of creating machines based on the laws of quantum mechanics instead of the laws of classical physics.

- 1994 - Peter Shor came up with a quantum algorithm to factor very large numbers in polynomial time.
- 1997 - Lov Grover develops a quantum search algorithm with O(√N) complexity
- 2018 – Google announces a 72 Qbit quantum chip



Mathematical Foundations

Slides are based on presentation by Andrew Helwer
https://www.youtube.com/watch?v=F_Riqjdh2oM&t=23s
and the book
Quantum Computing for Computer Scientists by N.S. Yanofsky and M.A. Mannucci, Cambridge University Press, 3rd edition, 2018



Representing classical bits as vectors

One bit with the value 0, also written as

Dirac Notation

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |0\rangle$$

One bit with the value 1, also written as

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad |1\rangle$$

Matrix Multiplication

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}$$

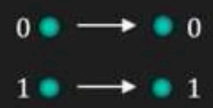
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} aw + by & ax + bz \\ cw + dy & cx + dz \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Mathematical Foundations

Operations on a single, classical bit (cbit)

Identity	$f(x) = x$		$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
Negation	$f(x) = \neg x$		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
Constant-0	$f(x) = 0$		$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
Constant-1	$f(x) = 1$		$\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$