Exercise 11 –Structs, Function and Operator Overloading

Informatik I für Mathematiker und Physiker (HS 2015)

Yeara Kozlov

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

HW #9 Feedback

- Question 1 - overall went well.

- Question 3

Assignment 3 – Skript-Aufgabe 126 (4 points)

In how many ways can you own CHF 1? Despite its somewhat philosophical appearance, the question is a mathematical one. Given some amount of money, in how many ways can you partition it using the available denominations (bank notes and coins)? Today's denominations in CHF are 1000, 200, 100, 50, 20, 10 (banknotes), 5, 2, 1, 0.50, 0.20, 0.10, 0.05 (coins). The amount of CHF 0.20, for example, can be owned in four ways (to get integers, let's switch to centimes): $(20), (10, 10), (10, 5, 5), (5, 5, 5, 5)$. The amount of CHF 0.04 can be owned in no way, while there is exactly one way to own CHF 0.00 (you cannot have 4 centimes in your wallet, but you *can* have no money at all in your wallet).

Solve the problem for a given input amount, by writing the following function (all values to be understood as centimes).

Agenda

- Const references
- Structs
- Function overloading
- Operator overloading
- Graded homework

Agenda

- **Const references**
- Structs
- Function overloading
- Operator overloading
- Graded homework

const references

- Variables cannot be changed via const references.

```
int a = 5;
int& b = a;
const int& c = a;

c++;  // error, a cannot be changed through const reference c
b++;  // a is now 6, a can still be changed through other non-const references
a++;  // a is now 7, a can still change through itself
```

- const references can be initialized with a r-value
  - For example: in function calls.
  - `const int & a = 5; //compiles`
- Enable a function to do call by value and call by reference

Agenda

- Const references
- **Structs**
- Function overloading
- Operator overloading
- Graded homework

Structs

- Definitions:

```
struct MyNewDataType {
        member_type1 mMember1;
        member_type1 mMember1;
        // …
};
```

- Variable declaration:

```
MyNewDataType myNewDataObject;
```

- Member assignment:

```
myNewDataObject.mMember1 = … ;
```

Structs - example

- Definitions:

```
struct Numbers {
        int i;
        float f;
};
```

- Variable declaration and initialization:

```
Numbers m1;
Numbers m2 = {5, 4.f};
Numbers m3, m4 = {1, 2.0f};
```

- Assignment:

```
m1 = m2; // m1.i=m2.i; m1.f=m2.f
```

Structs - member static arrays

- Structs can have static array members
- Static arrays will be copied with an assignment operator

```
struct strange {
  int n;
  bool b;
  int a[3];
};

int main () {
  strange x = {1, true, {1,2,3}};
  strange y = x;          // the array-member is also copied
  std::cout << y.n << "   " << y.a[2] << "\n"; // outputs: 1   3
  return 0;
}
```

Geometry Exercise

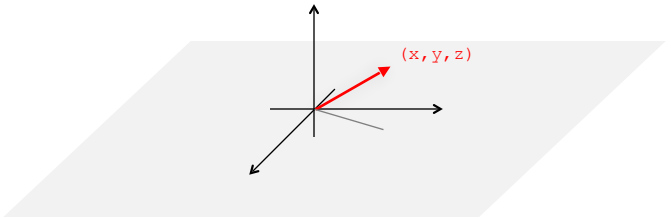
Similar to:
Old Exam Question Feb. 2010, Ex. 5

Exercise

In this exercise we will implement a representation of 3D-geometrical objects in a computer game.

Given is a struct `vec` which stores 3D-vectors.

```
struct vec {
  double x, y, z;
};
```



(Based on: Exam Feb. 2010, Ex. 5)

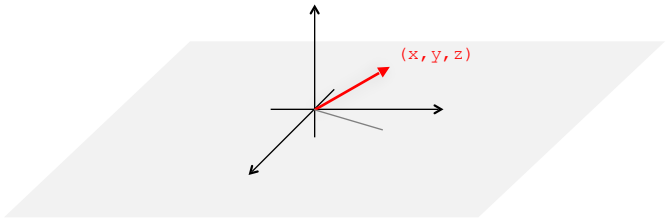
Exercise a)

Exercise a)

Implement the following function which computes a new vector obtained by adding two vectors.

```
// POST: returns the sum of two vectors
vec sum(const vec& a, const vec& b);
```

```
struct vec {
  double x, y, z;
};
```



(Based on: Exam Feb. 2010, Ex. 5)

Exercise a)

Solution a)

```
// POST: returns the sum of two vectors
vec sum(const vec& a, const vec& b) {
  vec tmp;
  tmp.x = a.x + b.x;
  tmp.y = a.y + b.y;
  tmp.z = a.z + b.z;
  return tmp;
}
```


(Based on: Exam Feb. 2010, Ex. 5)

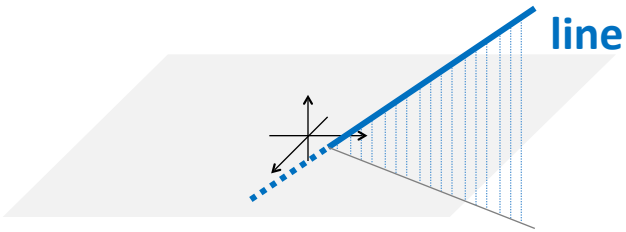
Exercise b)

Exercise b)

Propose a struct named `line`, which can be used to represent 3D-straight-lines.

A particular straight line does not have to be representable uniquely, but conversely every object of type `line` has to represent a unique straight line. If necessary you can for this reason define a suitable invariant (`// INV:...`) which has to be met when using the `line` struct.

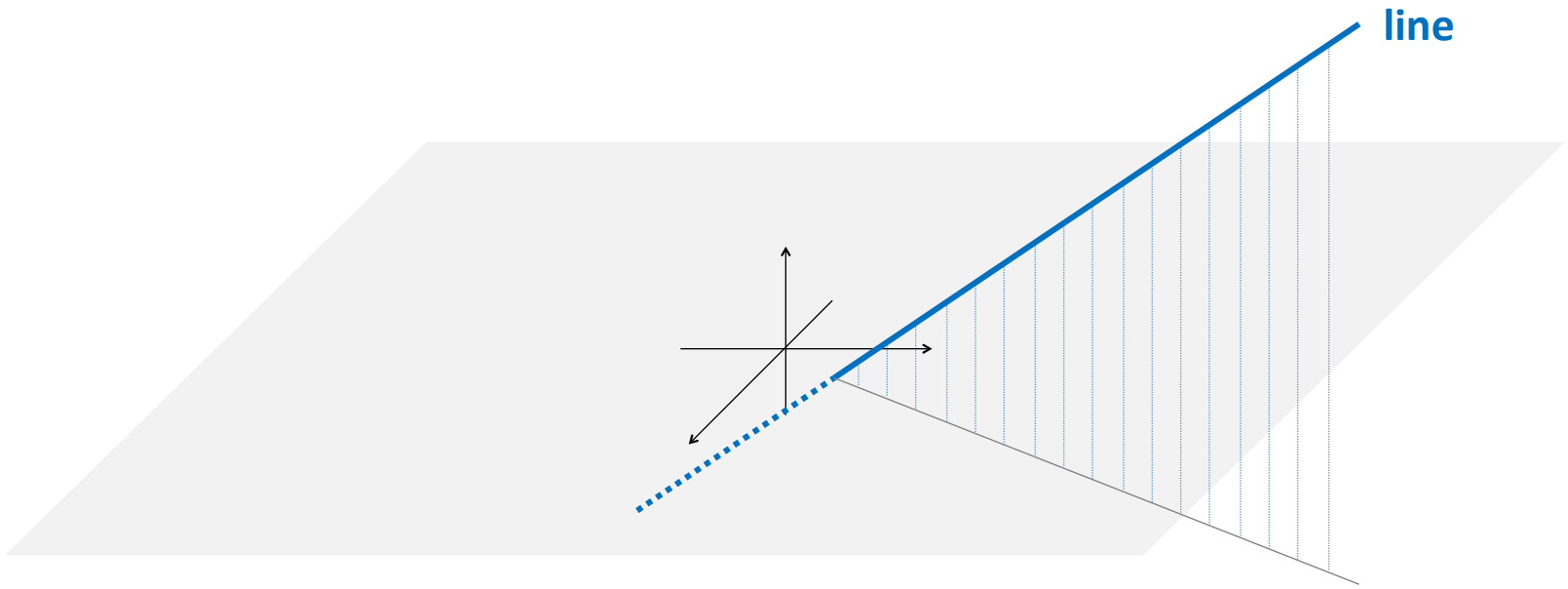
```
struct vec {
  double x, y, z;
};
```



(Based on: Exam Feb. 2010, Ex. 5)

Exercise b)

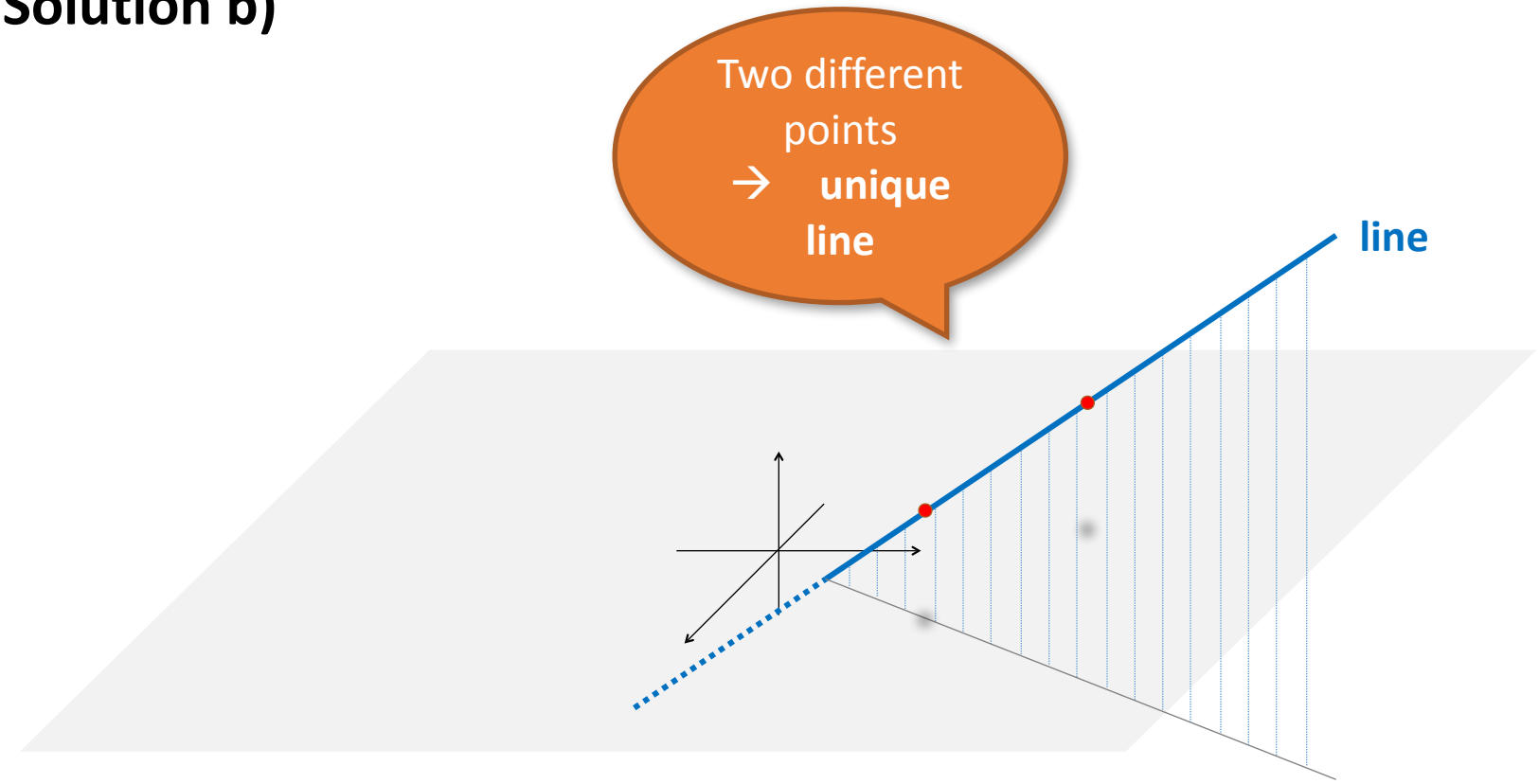
Solution b)



(Based on: Exam Feb. 2010, Ex. 5)

Exercise b)

Solution b)



(Based on: Exam Feb. 2010, Ex. 5)

Exercise b)

Solution b)

```
struct line {
  vec a, b; // INV: a != b
};
```


(Based on: Exam Feb. 2010, Ex. 5)

Exercise c)

Exercise c)

Based on your struct `line` implement the following function which returns a new shifted `line`.

```
// POST: returns a new line obtained by shifting l
//       by v.
line shift_line (const line& l, const vec& v);
```

```
struct vec {
  double x, y, z;
};
```

```
struct line {
  vec a, b; // INV: a != b
};
```


(Based on: Exam Feb. 2010, Ex. 5)

Exercise c)

Solution c)

```
// POST: returns a new line obtained by shifting l
//       by v.
line shift_line (const line& l, const vec& v) {
  line tmp;
  tmp.a = sum(l.a, v);
  tmp.b = sum(l.b, v);
  return tmp;
}
```


(Based on: Exam Feb. 2010, Ex. 5)

Agenda

- Const references
- Structs
- **Function overloading**
- Operator overloading
- Graded homework

Function Overloading

- The compiler identifies a function by its signature.
  - Function name
  - Number of parameters
  - Parameter types

➡ Two functions can have the same name, as long as the rest of their signature is different:

Number of parameters:

```
int f (int a) { ... }

int f (int a, int b) { ... }
```

Parameter type:

```
int f (int a) { ... }

int f (float a) { ... }
```

Function Overloading

- Different variable names **are not enough** for overloading

```
int f (int a) { ... }

int f (int b) { ... }  // compiler error
```

- Nor are return types

```
int f (int a) { ... }

double f (int a) { ... }  // compiler error
```

Function Overloading - Example

```
1  #include <iostream>
2
3  void out (const int i) {
4    std::cout << i << " (int)\n";
5  }
6
7  void out (const double i) {
8    std::cout << i << " (double)\n";
9  }
10
11 int main () {
12   out(3.5);          // 3.5 (double)
13   out(2);            // 2 (int)
14   out(2.0);          // 2 (double)
15   out(0);            // 0 (int)
16   out(0.0);          // 0 (double)
17   return 0;
18 }
```

Agenda

- Const references
- Structs
- Function overloading
- **Operator overloading**
- Graded homework

Operator Overloading

- Motivation: perform operations on structs naturally.
  - Examples: addition, subtraction, printing to std::cout

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

Overloading operators +=, -=, *=, /=

- One argument has to be a reference
- The return type has to be a reference
- Second argument is usually const

```
rational& operator+= (rational& a, const rational b) {
  a.n = a.n * b.d + a.d * b.n;
  a.d *= b.d;
  return a;
}
```

Overloading operators +, -, *, /

- Can be defined in terms of +=, -=, *= ….
  - Code reuse, less implementation, easier maintenance

```
rational operator+ (rational a, const rational b) {
  return a += b;
}
```

- Note differences in return and argument types

Overloading operators +, -, *, /

- Suppose we change the struct from:

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

- To:

```
struct rational {
  bool sign;
  unsigned int n;
  unsigned int d; // INV: d != 0
};
```

- Only need to change the +=, -= … operators.

Overloading operators >>, <<

- Convenient usage of iostreams

```
std::istream& operator>> (std::istream& in, rational& r) {
    char c;
    in >> r.n >> c >> r.d;
    return in;
}
```

- Left operand is a reference to the stream
- Return type is the input / output stream (by reference).
- >> operator inside the operator body is the normal std::cin operator.

Tribool Exercise

Tribool Exercise

- **Tribool:** three-valued logic
  `{false, unknown, true}`

(From: Script Exercise 142)

Tribool Exercise

- **Tribool:** three-valued logic
  `{false, unknown, true}`

- Operators `AND`, `OR` exist:

AND	false	unknown	true
false	false	false	false
unknown	false	unknown	unknown
true	false	unknown	true

OR	false	unknown	true
false	false	unknown	true
unknown	unknown	unknown	true
true	true	true	true

(From: Script Exercise 142)

Exercise a)

Exercise a)

Implement a type `Tribool` which will be used to represent variables for three-valued logic.

(Remember: `{false, unknown, true}`)

(From: Script Exercise 142)

Solution a)

Other solutions are of course also possible.

```
struct Tribool {
  // 0 = false, 1 = unknown, 2 = true
  unsigned int value; // INV: value in {0, 1, 2}
};
```

(This solution has handy properties for later subtasks.)

(From: Script Exercise 142)

Exercise b)

Exercise b)

Implement the boolean operators `&&` and `||` for your `Tribool` type.

&&	false	unknown	true
false	false	false	false
unknown	false	unknown	unknown
true	false	unknown	true

||	false	unknown	true
false	false	unknown	true
unknown	unknown	unknown	true
true	true	true	true

(From: Script Exercise 142)

Solution b)

Other solutions also possible.

But we can benefit from representation `{0,1,2}`.

&&	false	unknown	true
false	false	false	false
unknown	false	unknown	unknown
true	false	unknown	true

||	false	unknown	true
false	false	unknown	true
unknown	unknown	unknown	true
true	true	true	true

(From: Script Exercise 142)

Solution b)

Other solutions also possible.

But we can benefit from representation `{0,1,2}`.

&&	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

||	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

(From: Script Exercise 142)

Solution b)

Other solutions also possible.

But we can benefit from representation {0,1,2}

Use **minimum.**

&&	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

||	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

(From: Script Exercise 142)

Solution b)

Other solutions also possible.

But we can benefit from representation {0,1,2}

Use **minimum.**

&&	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

Use **maximum.**

||	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

(From: Script Exercise 142)

Solution b)

AND:

```
// POST: returns x AND y
Tribool operator&& (const Tribool x, const Tribool y) {
  Tribool result;
  result.value = std::min(x.value, y.value);
  return result;
}
```

OR:

```
// POST: returns x OR y
Tribool operator|| (const Tribool x, const Tribool y) {
  Tribool result;
  result.value = std::max(x.value, y.value);
  return result;
}
```


(From: Script Exercise 142)

Exercise c)

Exercise c)

Determine which overloads are called. And their values.

(1)
(2)
(3)

```
Tribool operator&& (const Tribool x, const Tribool y); // as before

Tribool operator&& (const Tribool x, const bool y) {
  Tribool y_as_tribool;
  y_as_tribool.value = 2*y; // trick:  2*false == 0  and  2*true == 2
  return x && y_as_tribool;
}

Tribool operator&& (const bool x, const Tribool y) {
  return y && x;
}
```

```
Tribool t; t.value = 1; // unknown
t && true;
t && t;
false && t;
false && true;
```

Exercise c)

Determine which overloads are called. And their values.

(1)
(2)
(3)

```
Tribool operator&& (const Tribool x, const Tribool y); // as before

Tribool operator&& (const Tribool x, const bool y) {
  Tribool y_as_tribool;
  y_as_tribool.value = 2*y; // trick:  2*false == 0  and  2*true == 2
  return x && y_as_tribool;
}

Tribool operator&& (const bool x, const Tribool y) {
  return y && x;
}
```

```
Tribool t; t.value = 1; // unknown
t && true;              // (2) (1)          value: unknown
t && t;
false && t;
false && true;
```

Exercise c)

Determine which overloads are called. And their values.

(1)
(2)
(3)

```
Tribool operator&& (const Tribool x, const Tribool y); // as before

Tribool operator&& (const Tribool x, const bool y) {
  Tribool y_as_tribool;
  y_as_tribool.value = 2*y; // trick:  2*false == 0  and  2*true == 2
  return x && y_as_tribool;
}

Tribool operator&& (const bool x, const Tribool y) {
  return y && x;
}
```

```
Tribool t; t.value = 1; // unknown
t && true;              // (2) (1)          value: unknown
t && t;                 // (1)              value: unknown
false && t;
false && true;
```

Exercise c)

Determine which overloads are called. And their values.

(1)
(2)
(3)

```
Tribool operator&& (const Tribool x, const Tribool y); // as before

Tribool operator&& (const Tribool x, const bool y) {
  Tribool y_as_tribool;
  y_as_tribool.value = 2*y; // trick:  2*false == 0  and  2*true == 2
  return x && y_as_tribool;
}

Tribool operator&& (const bool x, const Tribool y) {
  return y && x;
}
```

```
Tribool t; t.value = 1; // unknown
t && true;              // (2) (1)          value: unknown
t && t;                 // (1)              value: unknown
false && t;             // (3) (2) (1)      value: false
false && true;
```

Exercise c)

Determine which overloads are called. And their values.

(1)
(2)
(3)

```
Tribool operator&& (const Tribool x, const Tribool y); // as before

Tribool operator&& (const Tribool x, const bool y) {
  Tribool y_as_tribool;
  y_as_tribool.value = 2*y; // trick:  2*false == 0  and  2*true == 2
  return x && y_as_tribool;
}

Tribool operator&& (const bool x, const Tribool y) {
  return y && x;
}
```

```
Tribool t; t.value = 1; // unknown
t && true;              // (2) (1)          value: unknown
t && t;                 // (1)              value: unknown
false && t;             // (3) (2) (1)      value: false
false && true;          //  normal bool &&
                        //  value: false
```

Exercise d)

Exercise d)

Overload the output operator `<<` for your `Tribool` type:

```
// POST: Tribool value is written to o
std::ostream& operator<< (std::ostream& o, const Tribool x);
```

(Hint: You can think of `o` as `std::cout`.
In fact, `std::cout` is of type `std::ostream`.

This means that your overload allows you to write
`std::cout << my_tribool;`)

(From: Script Exercise 142)

Solution d)

Solution (a very compact form):

```
// POST: Tribool value is written to o
std::ostream& operator<< (std::ostream& o, const Tribool x) {
  if      (x.value == 0)  return o << "false  ";
  else if (x.value == 1)  return o << "unknown";
  else                    return o << "true   ";
}
```


(From: Script Exercise 142)

Solution d)

Solution (a very compact form):

```
// POST: Tribool value is written to o
std::ostream& operator<< (std::ostream& o, const Tribool x) {
  if      (x.value == 0)  return o << "false  ";
  else if (x.value == 1)  return o << "unknown";
  else                    return o << "true   ";
}
```

Remark:

This is `operator<<` for strings. (*)

(*) It is the one you're using whenever you output something: `std::cout << "Hello";`

(From: Script Exercise 142)

Verify Implementation

Verify Code

- Print truth table to test implementation:

```
// Truth Table for &&
Tribool f; f.value = 0; // false
Tribool u; u.value = 1; // unknown
Tribool t; t.value = 2; // true

std::cout << (f && f) << (f && u) << (f && t) << "\n"
          << (u && f) << (u && u) << (u && t) << "\n"
          << (t && f) << (t && u) << (t && t) << "\n";
```


(From: Script Exercise 142)

Verify Code

- Print truth table to test implementation:

```
// Truth Table for &&
Tribool f; f.value = 0; // false
Tribool u; u.value = 1; // unknown
Tribool t; t.value = 2; // true

std::cout << (f && f) << (f && u) << (f && t) << "\n"
          << (u && f) << (u && u) << (u && t) << "\n"
          << (t && f) << (t && u) << (t && t) << "\n";
```

```
Output is:

        false    false    false
        false    unknown  unknown
        false    unknown  true
```

(From: Script Exercise 142)

Agenda

- Const references
- Structs
- Function overloading
- Operator overloading
- **Graded homework**

Hex Instructions

- 1 hex digits = 4 bits = half a byte

`0xb000014`

```
opcode = 0xb
A = 0
B = 0
C= 0x14 = 20 (dec)
```

`0xc900000`

```
opcode = 0xc
A = 2
B = 1          (9 = 10 01)
C= 0
```

CPU Simulator

(Inspired by slides by Martin Bättig and Gian Ulli)

Example Program

- How would the processor execute this program?

```
0xb000002a
0xb4000007
0xd1000000
0xc9000000
0xe8000000
0x00000000
```

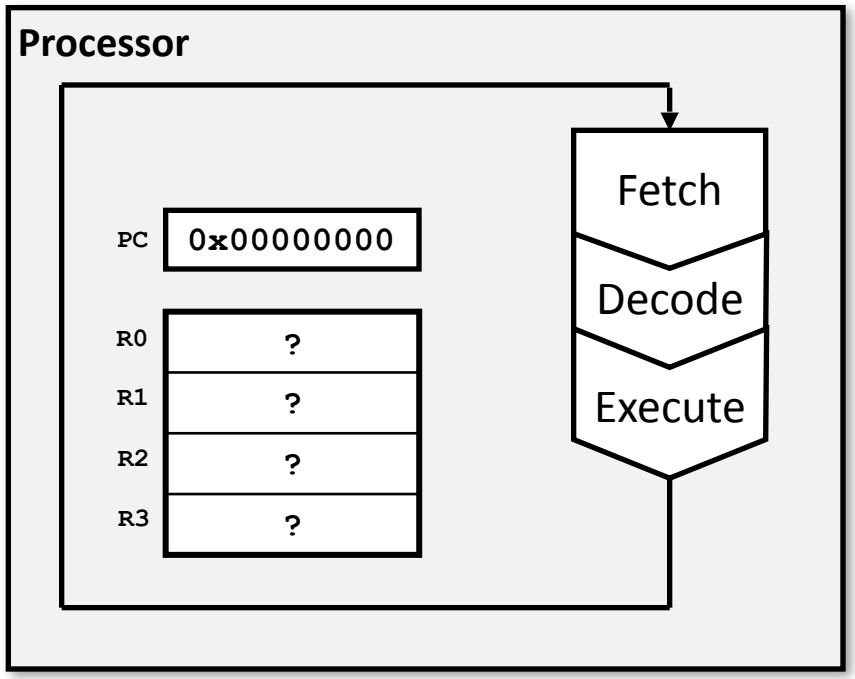
- Disassembly:

```
mov r0, 42     ;load 42 into r0
mov r1,  7     ;load 7 into r1
st  r0, r1     ;store r0 into mem[r1]
ld  r2, r1     ;load mem[r1] into r2
out r2,  0     ;output r2 as number
hlt            ;stop
```

(; denote comments in assembly)

CPU Simulator

Initial State



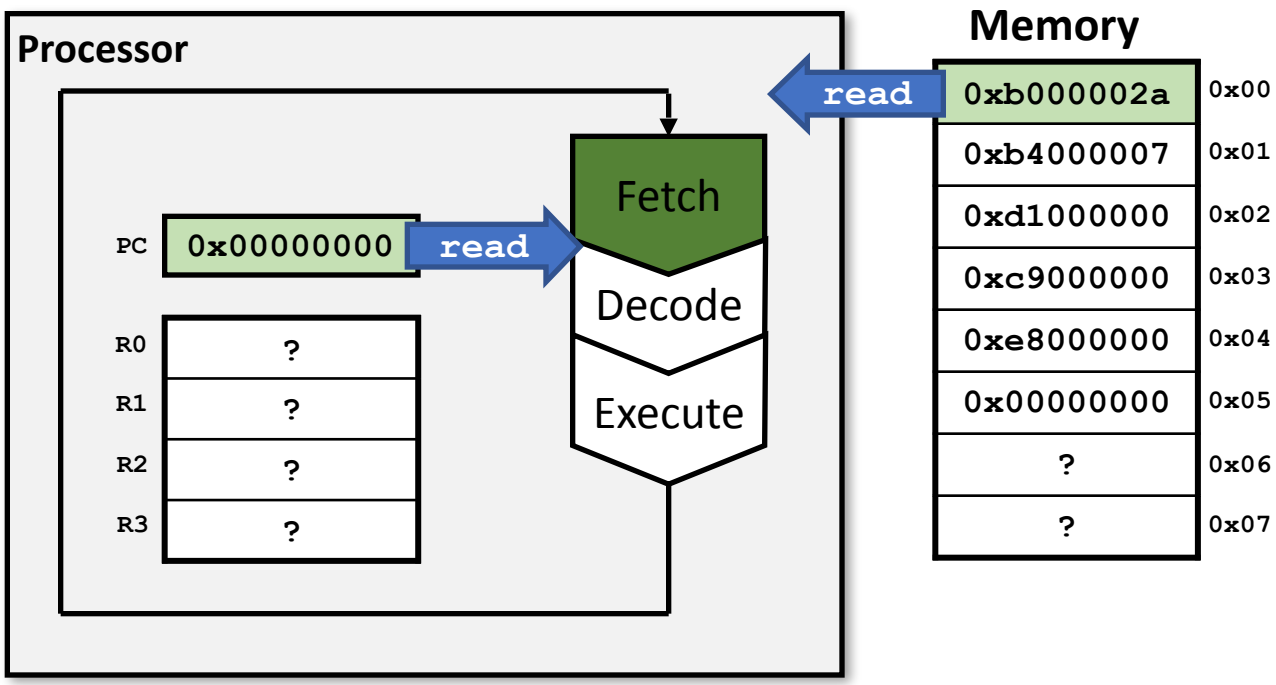
Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
?	0x07

For simplicity: Memory with just 8 adresses instead of 2^{16}.

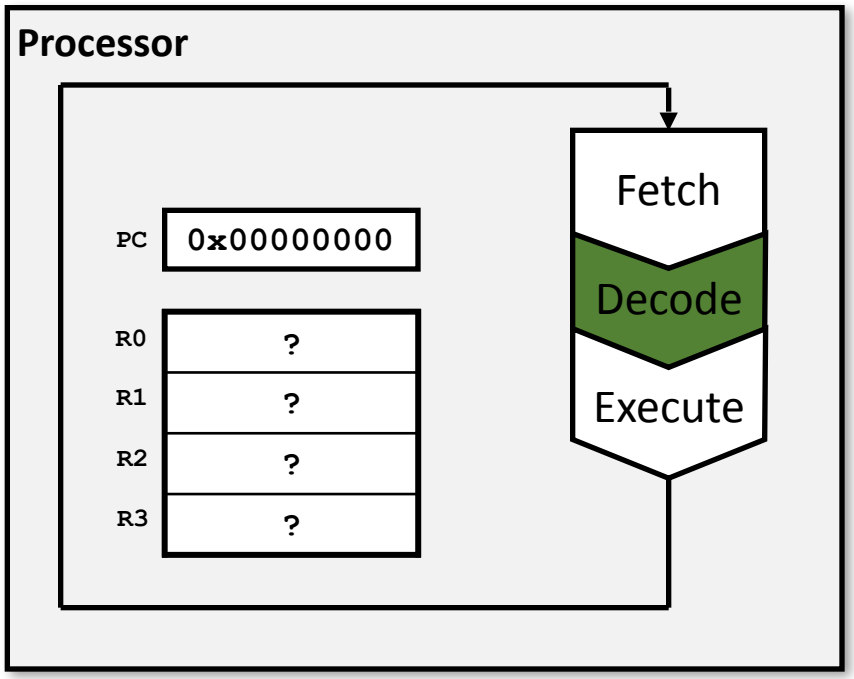
CPU Simulator

```
inst: 0xb000002a
```



For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
?	0x07

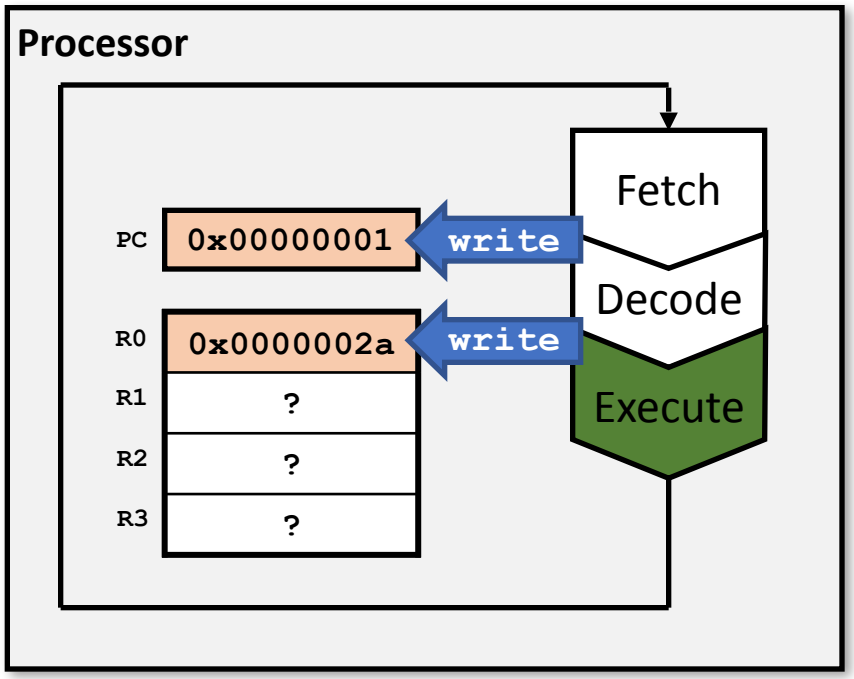
```
inst: 0xb000002a
Opcode: 0xb [mov]
Op A: 0x0
Op B: 0x0
Op C: 0x00002a
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator

`mov`: Load the (unsigned) constant `c` into register `reg[A]`.

```
inst: 0xb000002a
Opcode: 0xb [mov]
Op A: 0x0
Op B: 0x0
Op C: 0x00002a
```

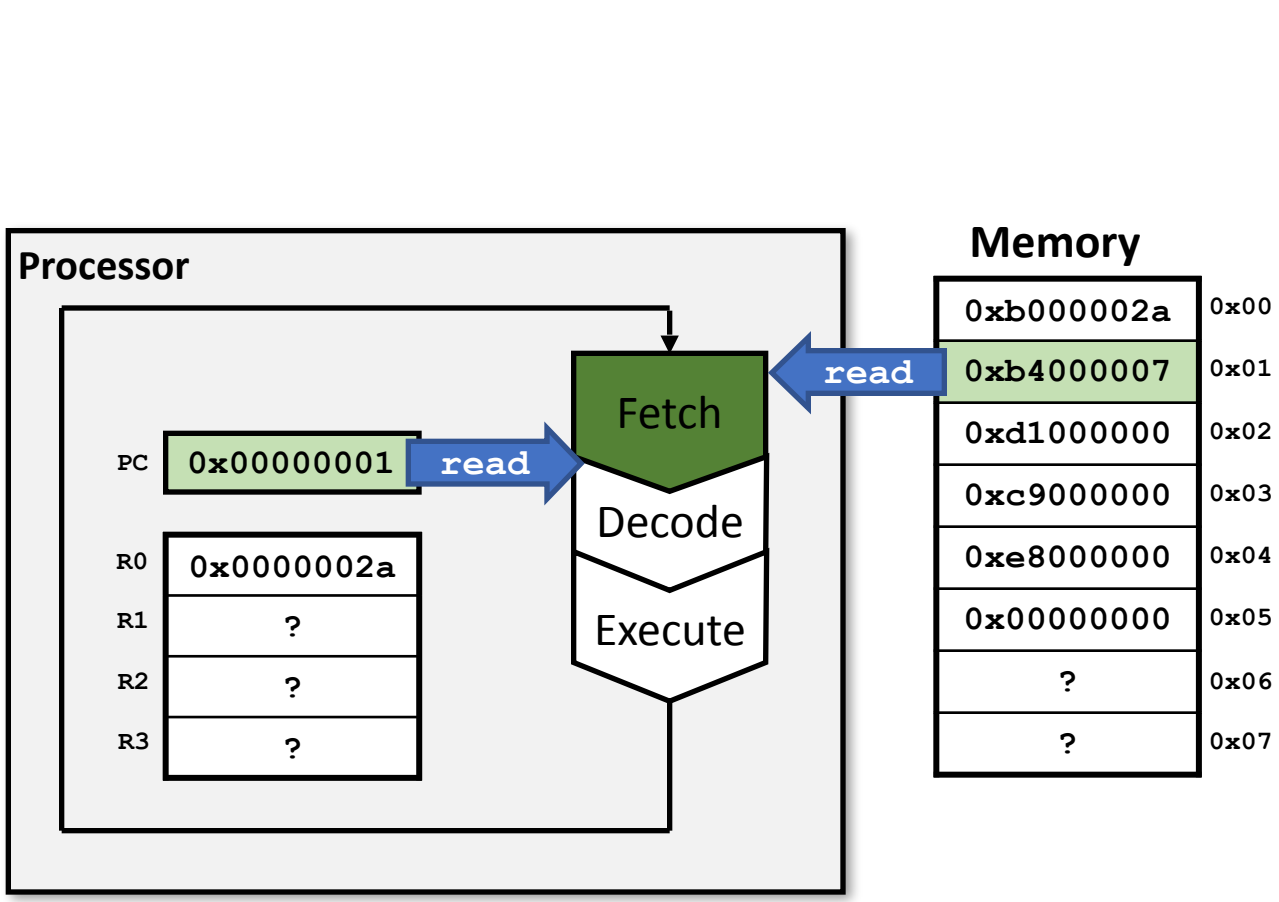


Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
?	0x07

For simplicity: Memory with just 8 adresses instead of 2^{16}.

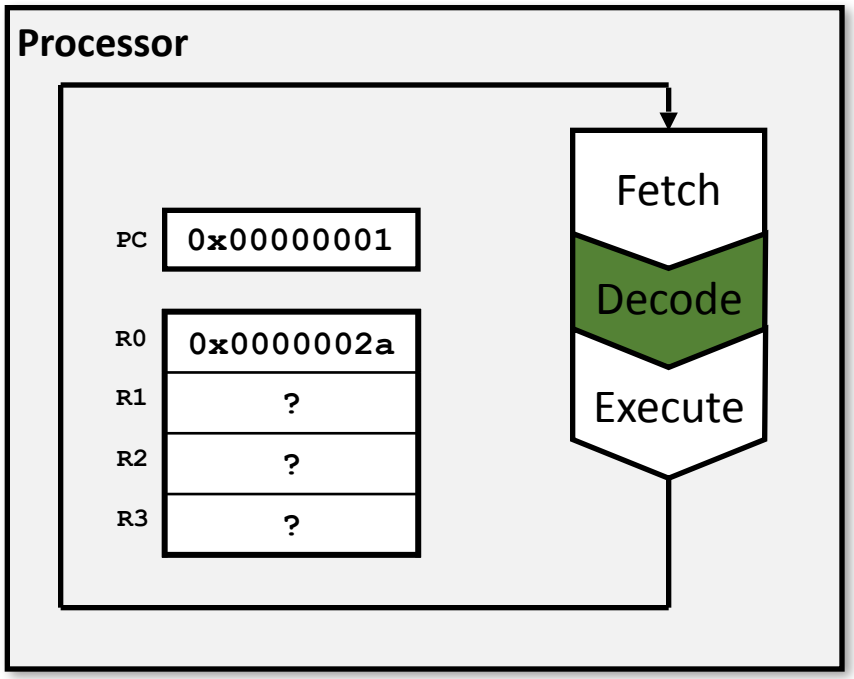
CPU Simulator




```
inst: 0xb4000007
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
?	0x07

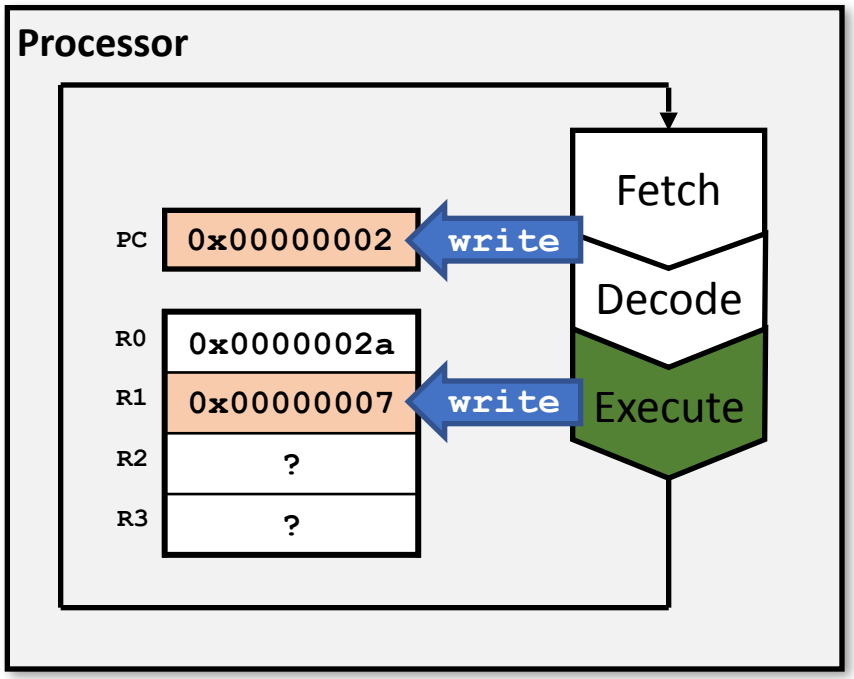
```
inst: 0xb4000007
Opcode: 0xb [mov]
Op A: 0x1
Op B: 0x0
Op C: 0x000007
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator

`mov`: Load the (unsigned) constant `c` into register `reg[A]`.

```
inst: 0xb4000007
Opcode: 0xb [mov]
Op A: 0x1
Op B: 0x0
Op C: 0x000007
```

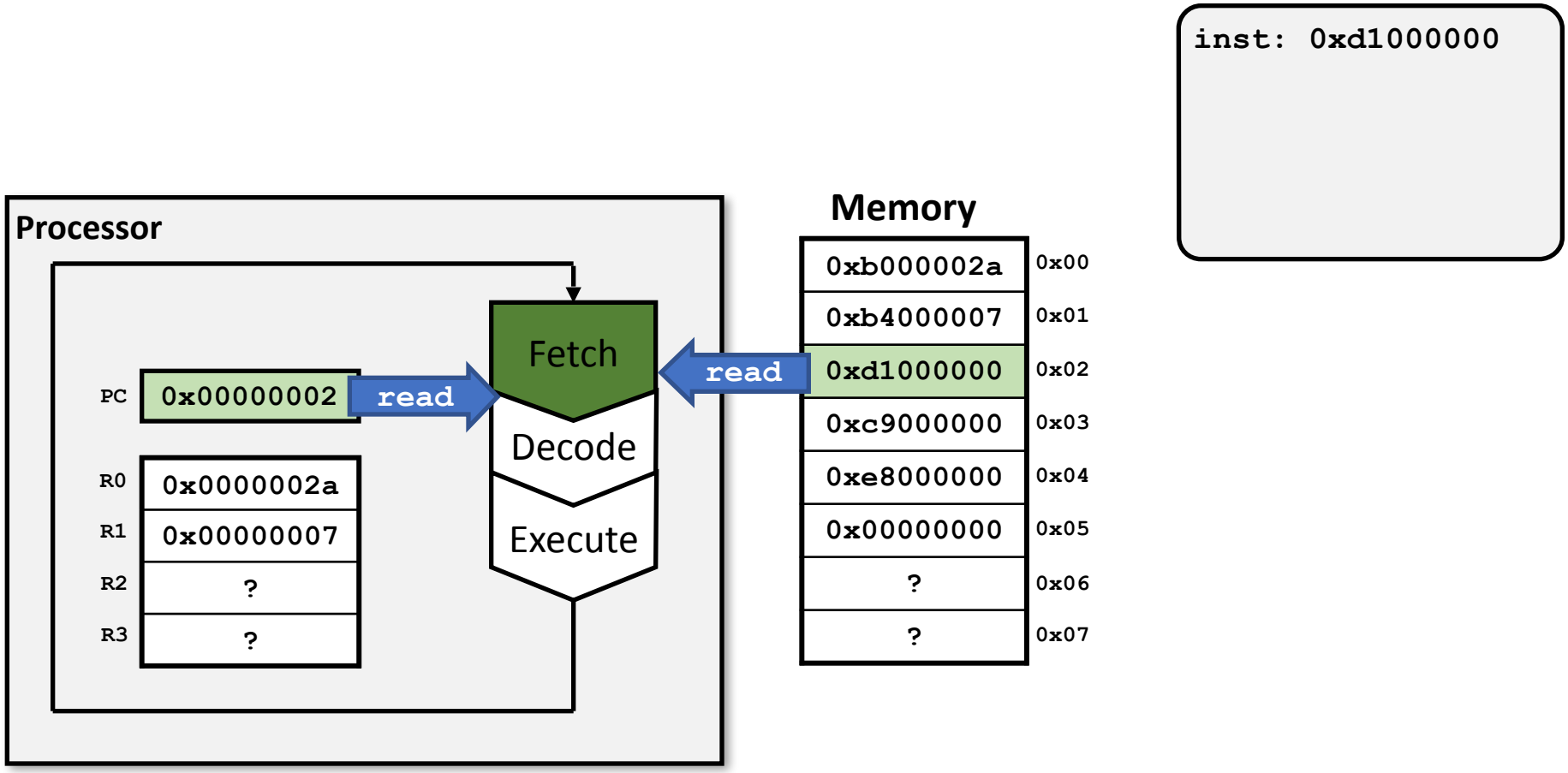


Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
?	0x07

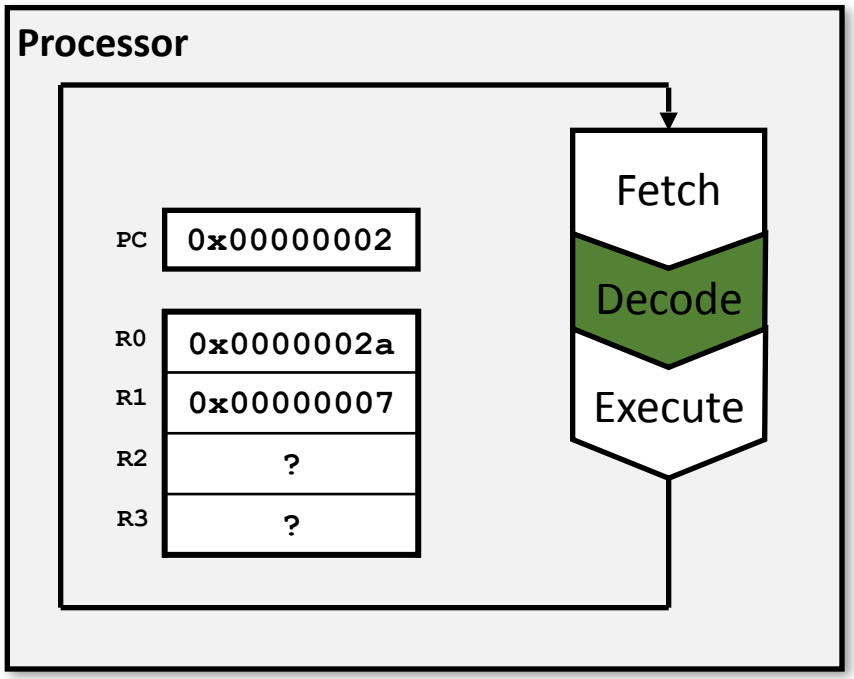
For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
?	0x07

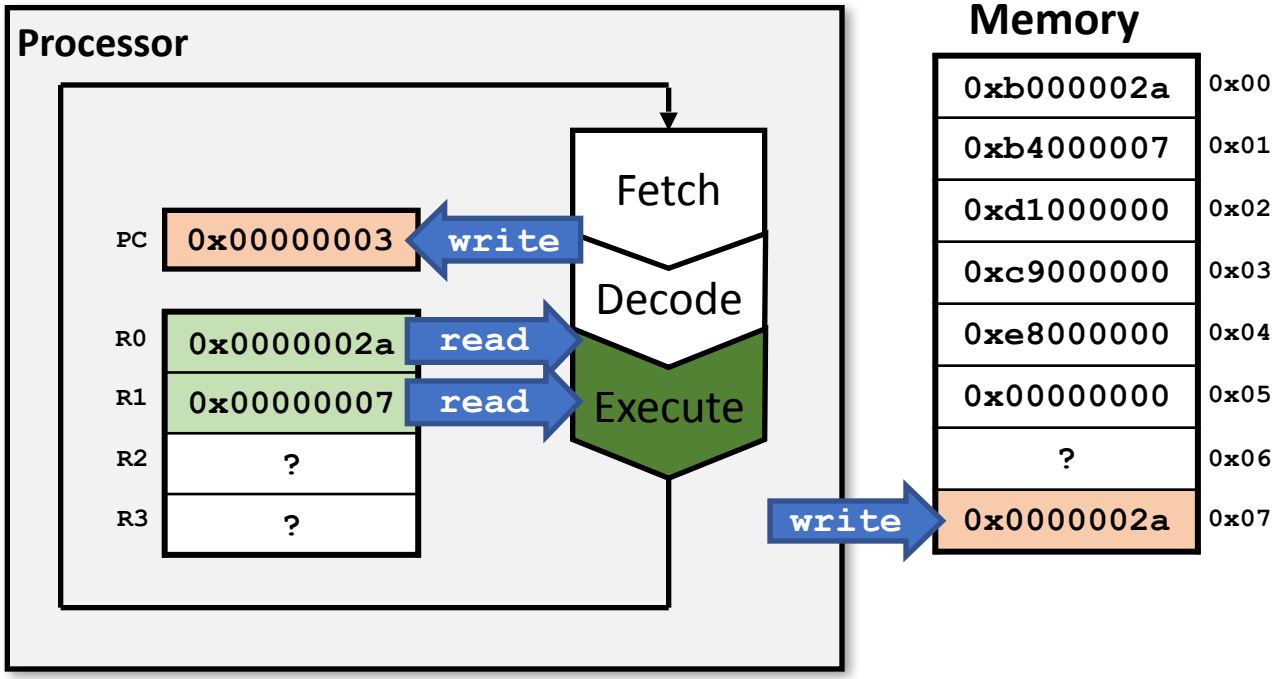
```
inst: 0xd1000000
Opcode: 0xd [st]
Op A: 0x0
Op B: 0x1
Op C: 0x000000
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator

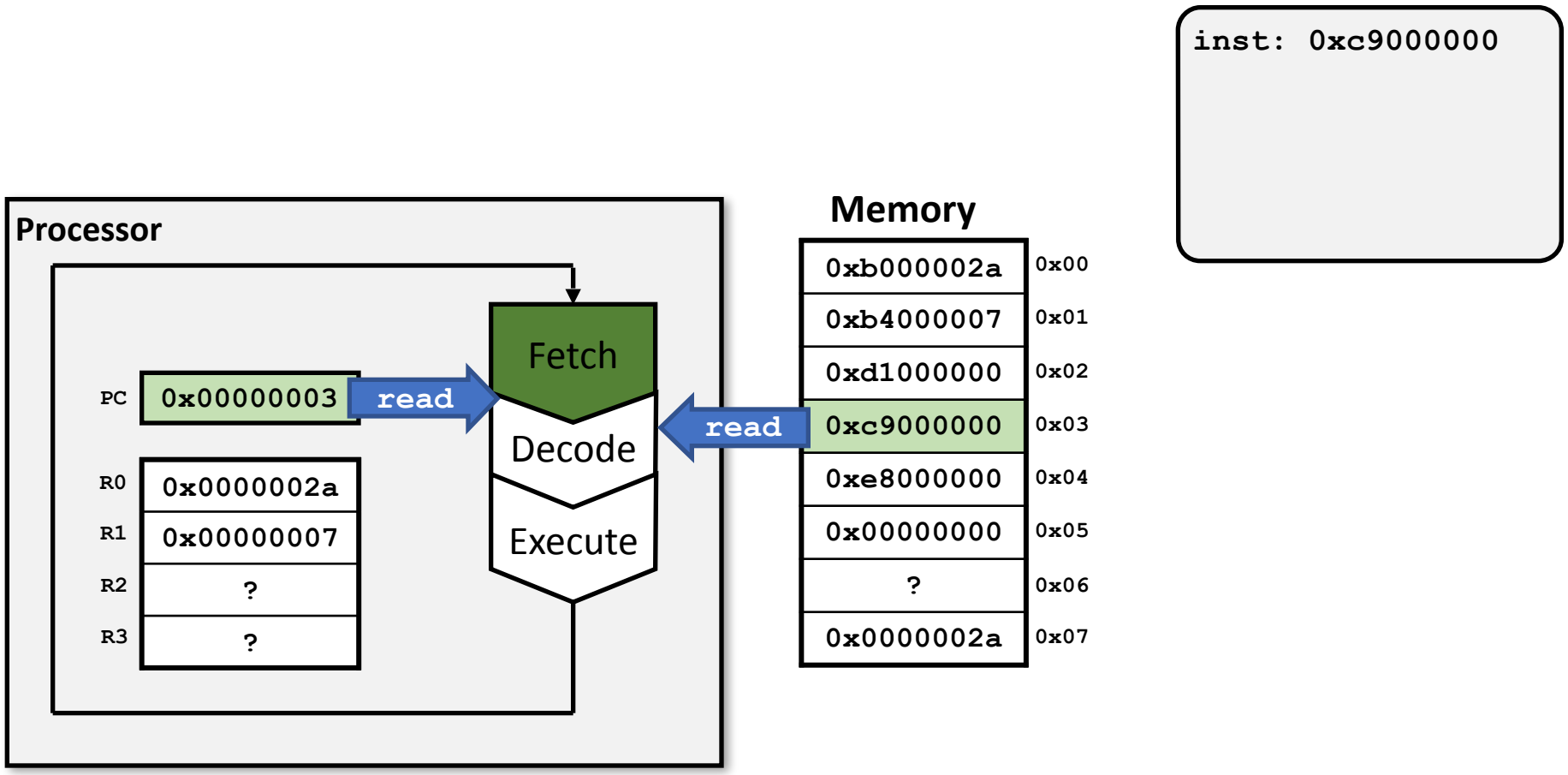
`st:` Store the value of `reg[A]` into `mem[reg[B]]`.

```
inst: 0xd1000000
Opcode: 0xd [st]
Op A: 0x0
Op B: 0x1
Op C: 0x000000
```



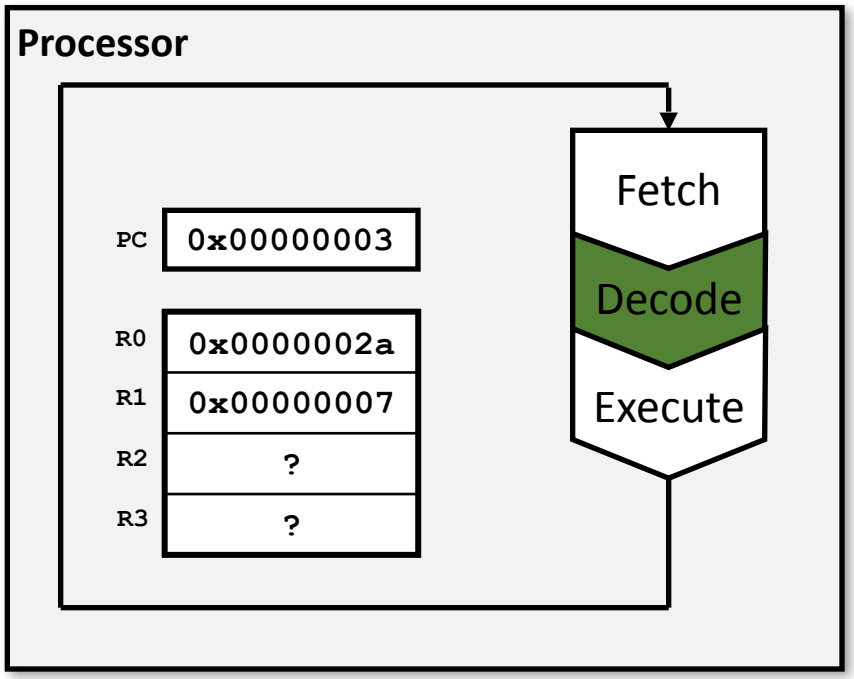
For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
0x0000002a	0x07

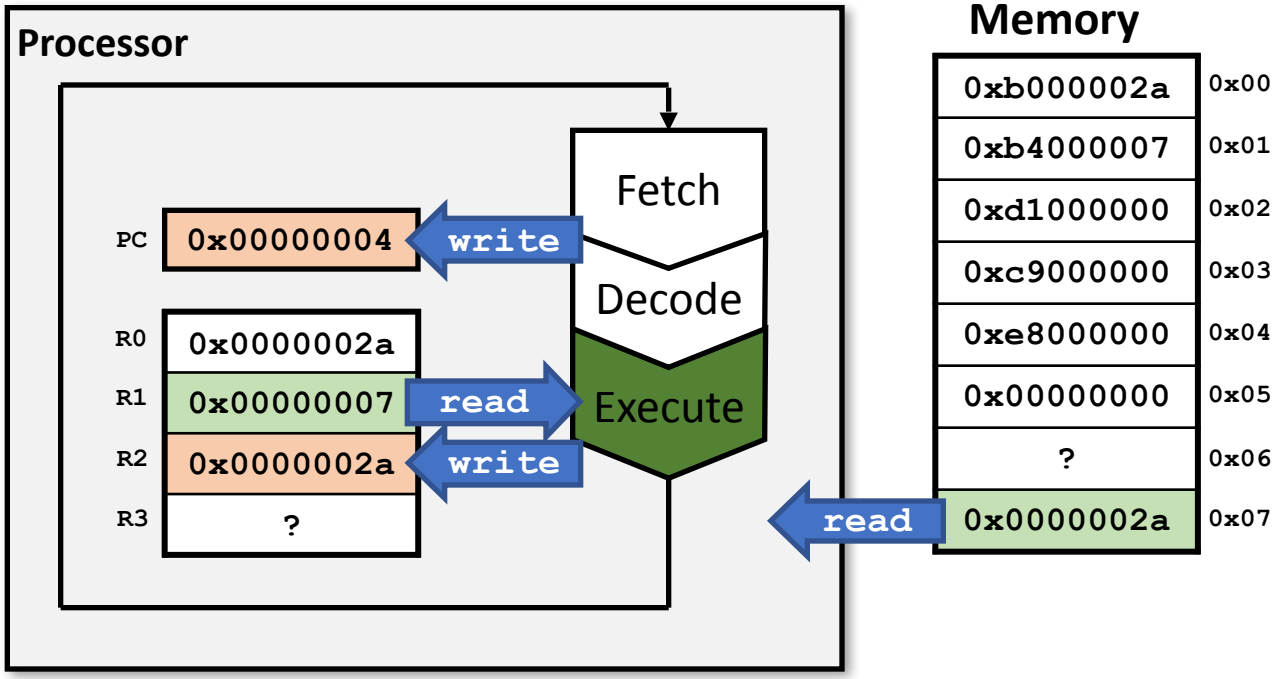
```
inst: 0xc9000000
Opcode: 0xc [ld]
Op A: 0x2
Op B: 0x1
Op C: 0x000000
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator

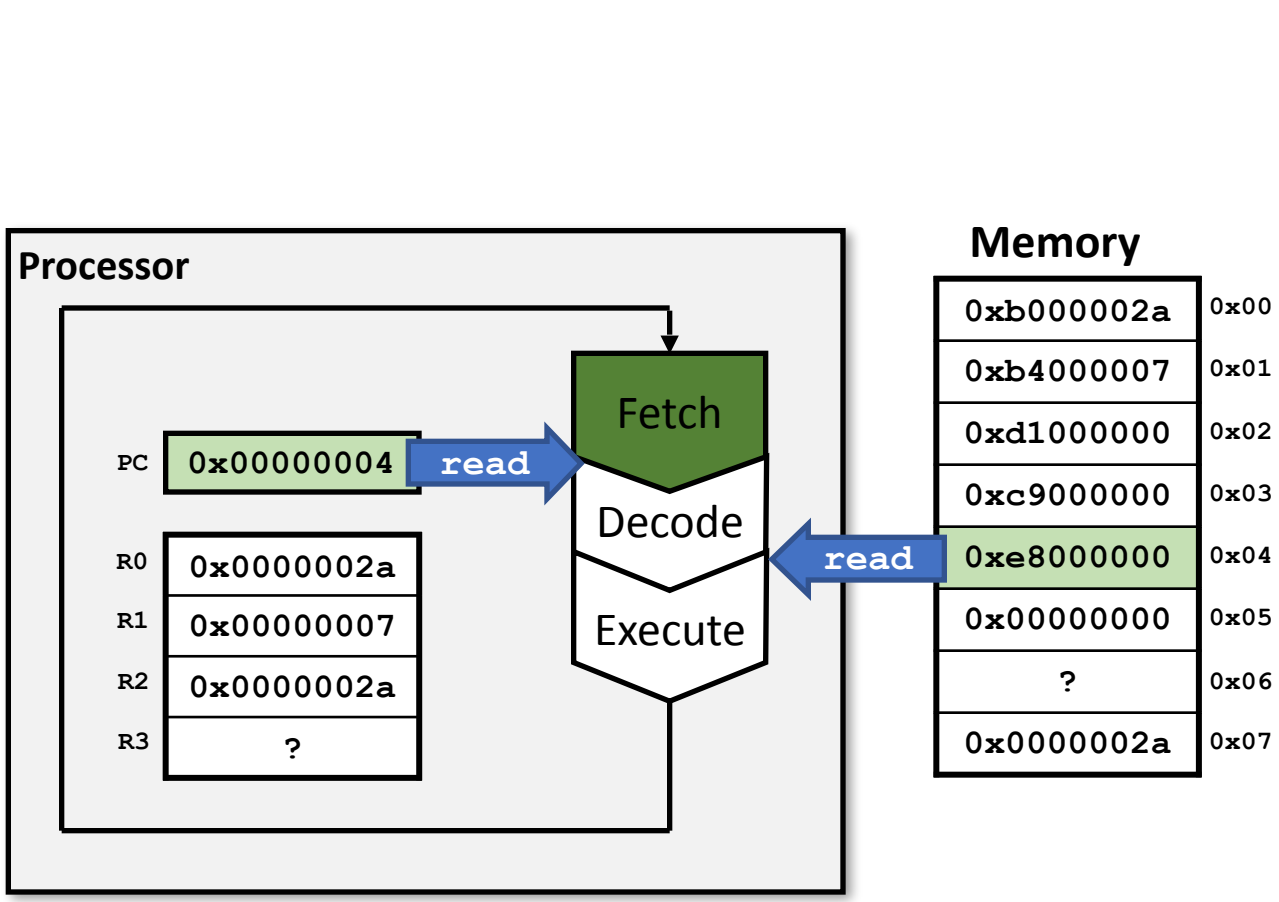
`ld:` Load the value stored at `mem[reg[B]]` into `reg[A]`.

```
inst: 0xc9000000
Opcode: 0xc [ld]
Op A: 0x2
Op B: 0x1
Op C: 0x000000
```



For simplicity: Memory with just 8 adresses instead of 2^{16}.

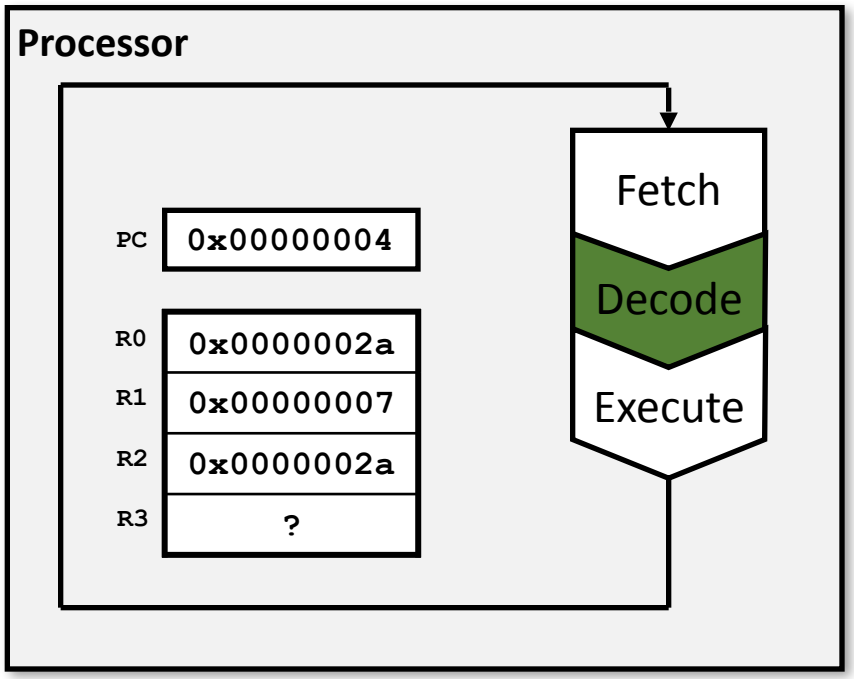
CPU Simulator




```
inst: 0xe8000000
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
0x0000002a	0x07

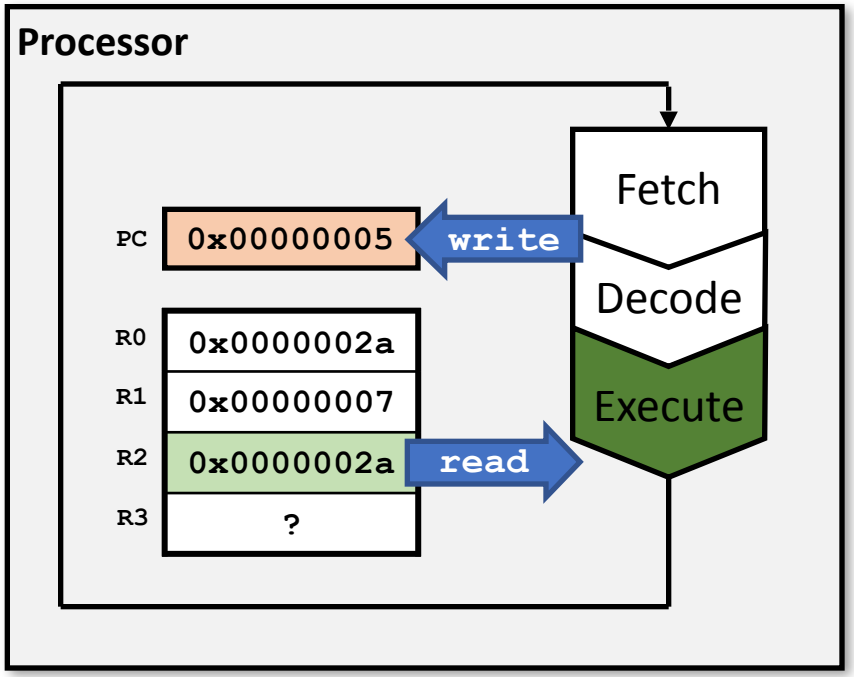
```
inst: 0xe8000000
Opcode: 0xe [out]
Op A: 0x2
Op B: 0x0
Op C: 0x000000
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator

`out:` Write the value stored at `reg[A]` to standard out `std::cout`. If `c = 0` then write as 32-bit unsigned decimal integer else write as character.

```
inst: 0xe8000000
Opcode: 0xe [out]
Op A: 0x2
Op B: 0x0
Op C: 0x000000
```



Memory

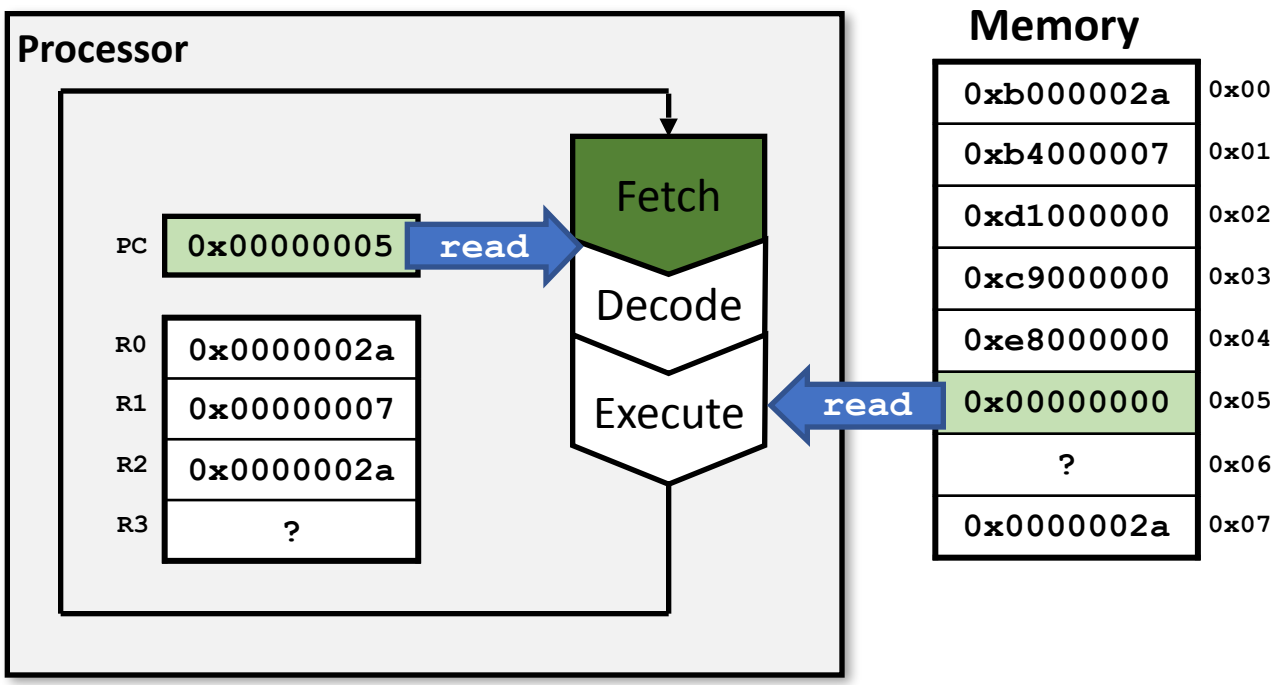
0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
0x0000002a	0x07

Output: 42

For simplicity: Memory with just 8 adresses instead of 2^{16}.

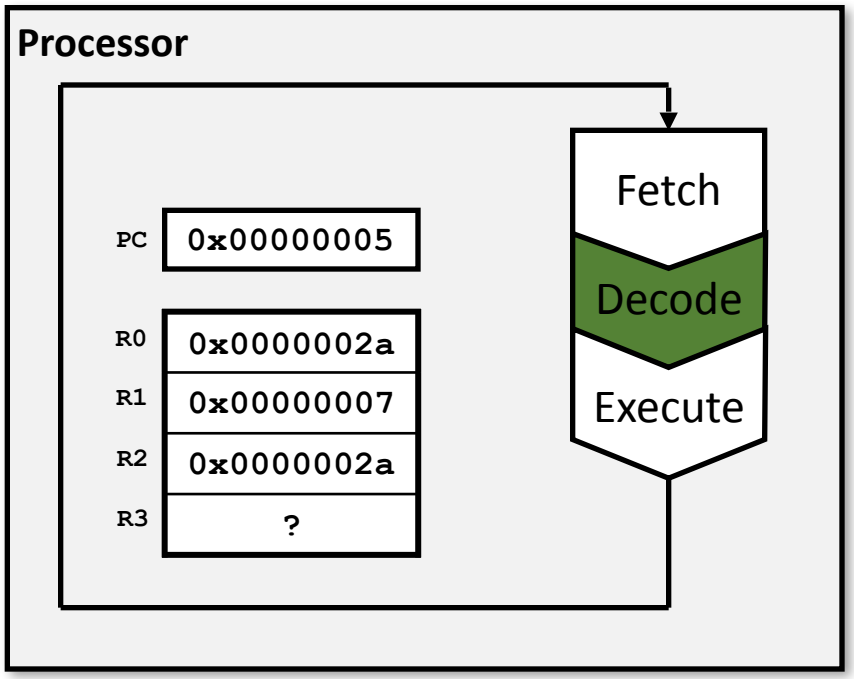
CPU Simulator

```
inst: 0x00000000
```



For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator



Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
0x0000002a	0x07

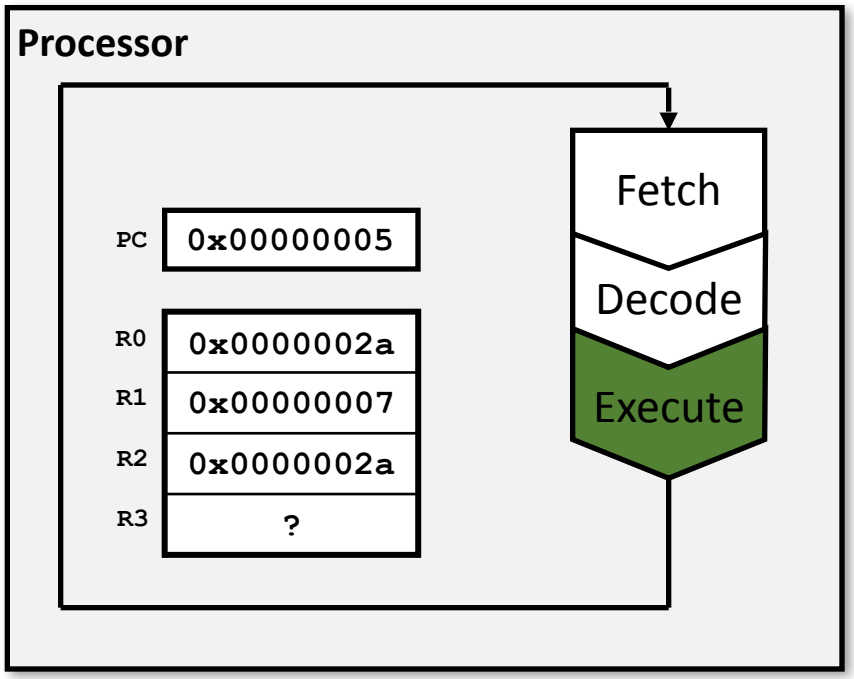
```
inst: 0x00000000
Opcode: 0x0 [hlt]
Op A: 0x0
Op B: 0x0
Op C: 0x000000
```

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator

`hlt`: Halts the system.

```
inst: 0x00000000
Opcode: 0x0 [hlt]
Op A: 0x0
Op B: 0x0
Op C: 0x000000
```



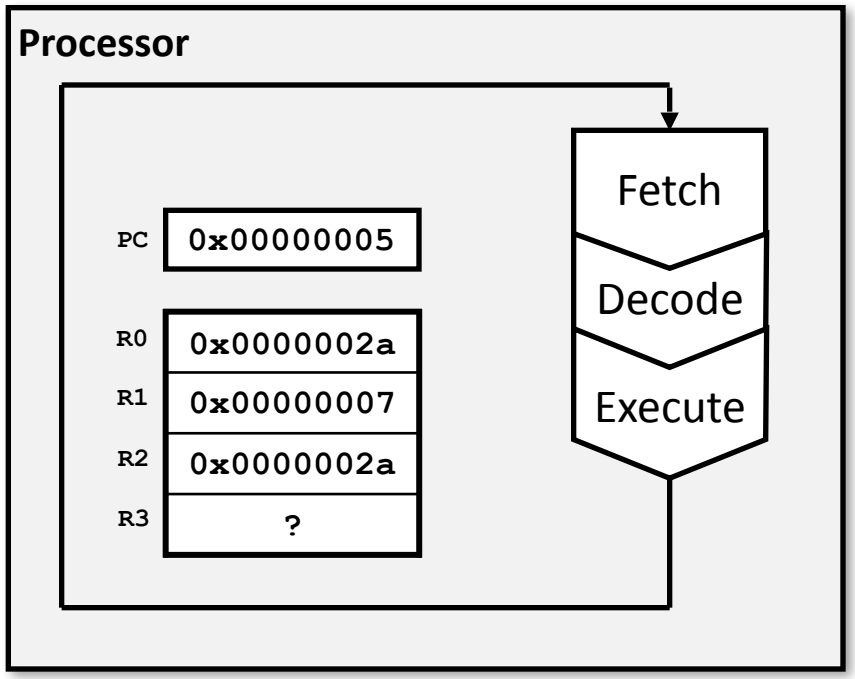
Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
0x0000002a	0x07

For simplicity: Memory with just 8 adresses instead of 2^{16}.

CPU Simulator

Simulation finished



Memory

0xb000002a	0x00
0xb4000007	0x01
0xd1000000	0x02
0xc9000000	0x03
0xe8000000	0x04
0x00000000	0x05
?	0x06
0x0000002a	0x07

For simplicity: Memory with just 8 adresses instead of 2^{16}.