

# INTRO TO COMPILER DEVELOPMENT

LOGAN CHIEN

<http://slide.logan.tw/compiler-intro/>

# LOGAN CHIEN

Software engineer at MediaTek

Ametuar LLVM/Clang developer

Integrated LLVM/Clang into Android NDK

# WHY I LOVE COMPILERS?

I have been a faithful reader of **Jserv's** blog for ten years.

I was inspired by the **compiler** and the **virtual machine** technologies mentioned in his blog.

I have decided to choose *compiler technologies* as my research topic since then.

# **UNDERGRADUATE COMPILER COURSE**

I took the undergraduate compiler course when I was a sophomore.

# MY PROFESSOR SAID ...

“We took many lectures to discuss about the parser.

However, when people say they are doing **compiler research**, with large possibility, they are *not* referring to the parsing technique.”

# I AM HERE TO ...

**Re-introduce** the compiler technologies,

Give a *lightening talk* on **industrial-strength** compiler design,

Explain the **connection** between compiler technologies and the industry.

# AGENDA

- Re-introduction to Compiler (30min)
- Industrial-strength Compiler Design (90min)
- Compiler and ICT Industry (20min)

# **RE-INTRODUCTION TO COMPILER**



# WHAT IS A COMPILER?

Compiler are tools for programmers to translate programmer's **thought** into computer runnable **programs**.

ANALOGY — *Translators* who turn from one language to another, such as those who translate Chinese to English.

**WHAT HAVE WE LEARNED IN  
UNDERGRADUATE COMPILER  
COURSE?**

# LEXER

Reads the input source code (as a sequence of bytes) and converts them into a stream of **tokens**.

```
unsigned background(unsigned foreground) {  
    if ((foreground >> 16) > 0x80) {  
        return 0;  
    } else {  
        return 0xffffffff;  
    }  
}
```

unsigned	background	(	unsigned	foreground	)	{	if	(	(	foreground	>>					
16	)	>	128	)	{	return	0	;	}	else	{	return	16777215	;	}	}

# PARSER

Reads the tokens and build an **AST** according to the syntax.

unsigned background ( unsigned foreground ) { if ( ( foreground >>  
16 ) > 128 ) { return 0 ; } else { return 16777215 ; } }

```
(procedure background
  (args '(foreground))
  (compound-stmt
    (if-stmt
      (bin-expr GE (bin-expr RSHIFT foreground 16) 128)
      (return-stmt 0)
      (return-stmt 16777215))))
```

# CODE GENERATOR

Generate the **machine code** or (assembly) according to the AST. In the undergraduate course, we usually simply do *syntax-directed translation*.

```
(procedure background
  (args '(foreground))
  (compound-stmt
    (if-stmt
      (bin-expr GE (bin-expr RSHIFT foreground 16) 128)
      (return-stmt 0)
      (return-stmt 16777215))))
```

```
    lsr w8, w0, #16
    cmp w8, #128
    b.lo .Lelse
    mov w0, wzr
    ret
.Lelse:
    orr w0, wzr, #0xffffffff
    ret
```

# WHAT'S MISSING?

Can a person who can only lex and parse sentences translate articles well?

# COMPILER REQUIREMENTS

A compiler should translate the source code precisely.

A compiler should utilize the device efficiently.

# THREE RELATED FIELDS

Programming Language

Computer Architecture

Compiler



# PROGRAMMING LANGUAGE THEORY

Essential component of a programming language: **type theory, variable scoping, language semantics**, etc.

How do people reason and compose a program?

Create an abstraction that is understandable to human and tracable to computers.

# EXAMPLE: SUBTYPE AND MUTABLE RECORDS

Why you can't perform following conversion in C++?

```
void test(int *ptr) {  
    int **p = &ptr;  
    const int** a = p; // Compiler gives warning  
    // ...  
}
```

This is related to covariant type and contravariance type. With PLT, we know that we can only choose two of (a) **covariant type**, (b) **mutable records**, and (c) **type consistency**.

```
void test(int *ptr) {  
    const int c = 0;  
    int **p = &ptr;  
    const int** a = p; // If it is allowed, bad programs will pass.  
    *a = &c;  
    *p = 5; // No warning here.  
}
```

# EXAMPLE: DYNAMIC SCOPING IN BASH

```
#!/bin/sh
v=1                # Initialize v with 1
foo () {
    echo "foo:v=${v}" # Which v is referred?
    v=2              # Which v is assigned?
}
bar () {
    local v=3
    foo
    echo "bar:v=${v}" # What will be printed?
}
v=4                # Assign 4 to v
bar
echo "v=${v}"      # What will be printed?
```

Ans: **foo:v=3**, **bar:v=2**, **v=4**. Surprisingly, *foo* is accessing local *v* in *bar* instead of the global *v*.

# EXAMPLE: NON-LEXICAL SCOPING IN JAVASCRIPT

```
// Javascript, the bad part
function bad(v) {
    var sum;
    with (v) {
        sum = a + b;
    }
    return sum;
}

console.log(bad({a: 5, b: 10}));
console.log(bad({a: 5, b: 10, sum: 100}));
```

Ans: The second `console.log()` prints undefined.

# COMPUTER ARCHITECTURE

Instruction set architecture: CISC vs. RISC.

Out-of-Order Execution vs. Instruction Scheduling.

Memory hierarchy

Memory model

# QUIZ: DO YOU REALLY KNOW C?

Is it guaranteed that **v** will always be loaded after **pred**?

```
int pred;  
int v;  
  
int get(int a, int b) {  
    int res;  
    if (pred > 0) {  
        res = v * a - v / b;  
    } else {  
        res = v * a + v / b;  
    }  
    return res;  
}
```

Ans: No. Independent reads/writes can be reordered. The standard only requires the result should be the same as running from top to bottom (in a single thread.)

# COMPILER ANALYSIS

**Data-flow analysis** — Analyze value ranges, check the conditions or constraints, figure out modifications to variables, etc.

**Control-flow analysis** — Analyze the structure of the program, such as *control dependency* and *loop structure*.

**Memory dependency analysis** — Analyze the memory access pattern of the access to array elements or pointer dereferences, e.g. *alias analysis*.

# ALIAS ANALYSIS

Determine whether two pointers can refer to the same object or not.

```
void move(char *dst, const char *src, int n) {  
    for (int i = 0; i < n; ++i) {  
        dst[i] = src[i];  
    }  
}
```

```
int sum(int *ptr, const int *val, int n) {  
    int res = 0;  
    for (int i = 0; i < n; ++i) {  
        res += *val;  
        *ptr++ = 10;  
    }  
    return res;  
}
```



# QUIZ: DO YOU KNOW C++?

```
class QMutexLocker {
public:
    union {
        QMutex *mtx_;
        uintptr_t val_;
    };
    void unlock() {
        if (val_) {
            if ((val_ & (uintptr_t)1) == (uintptr_t)1) {
                val_ &= ~(uintptr_t)1;
                mtx_>unlock();
            }
        }
    }
};
```

Pitfall: Reading from union fields that were not written previously results in undefined behavior. **Type-Based Alias Analysis (TBAA)** exploits this rule.

# COMPILER OPTIMIZATION

**Scalar optimization** — Fold the constants, remove the redundancies, change expressions with identities, etc.

**Vector optimization** — Convert several scalar operations into one vector operation, e.g. combining for add instruction into one vector add.

**Interprocedural optimization** — function inlining, devirtualization, cross-function analysis, etc.

# OTHER COMPILER-RELATED TECHNOLOGY

Just-in-time compilers

Binary translators

Program profiling and performance measurement

Facilities to run compiled executables, e.g. garbage  
collectors

# **INDUSTRIAL- STRENGTH COMPILER DESIGN**

What's the difference between *your final project* and the industrial-strength compiler?

# KEY DIFFERENCE

**Analysis** — Reasons program structures and changes of values.

**Optimization** — Applies several provably correct transformation which should make program run faster.

**Intermediate Representation** — Data structure on which analyses and optimizations are based.

# INTERMEDIATE REPRESENTATION<sup>1/4</sup>

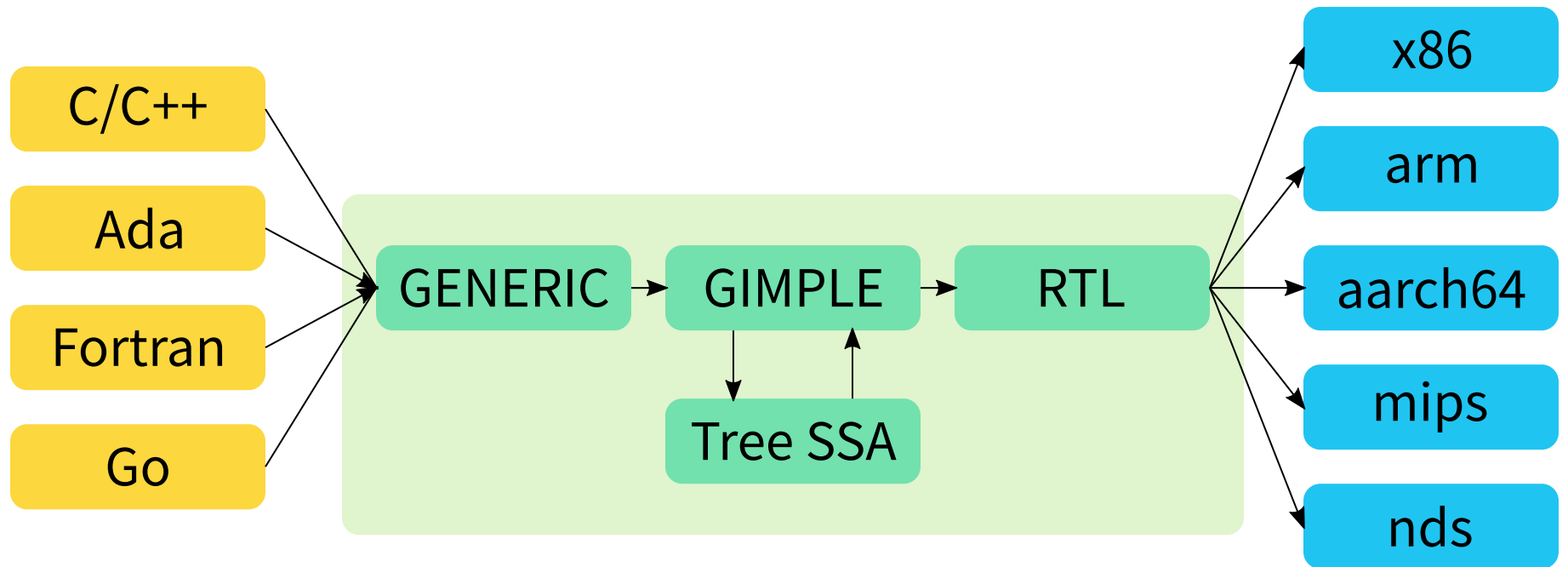
A **data structure** for program analyses and optimizations.

High-level enough to capture *important properties* and encapsulates hardware limitation.

Low-level enough to be *analyzed* by analyses and *manipulated* by transformations.

An **abstraction layer** for multiple front-ends and back-ends.

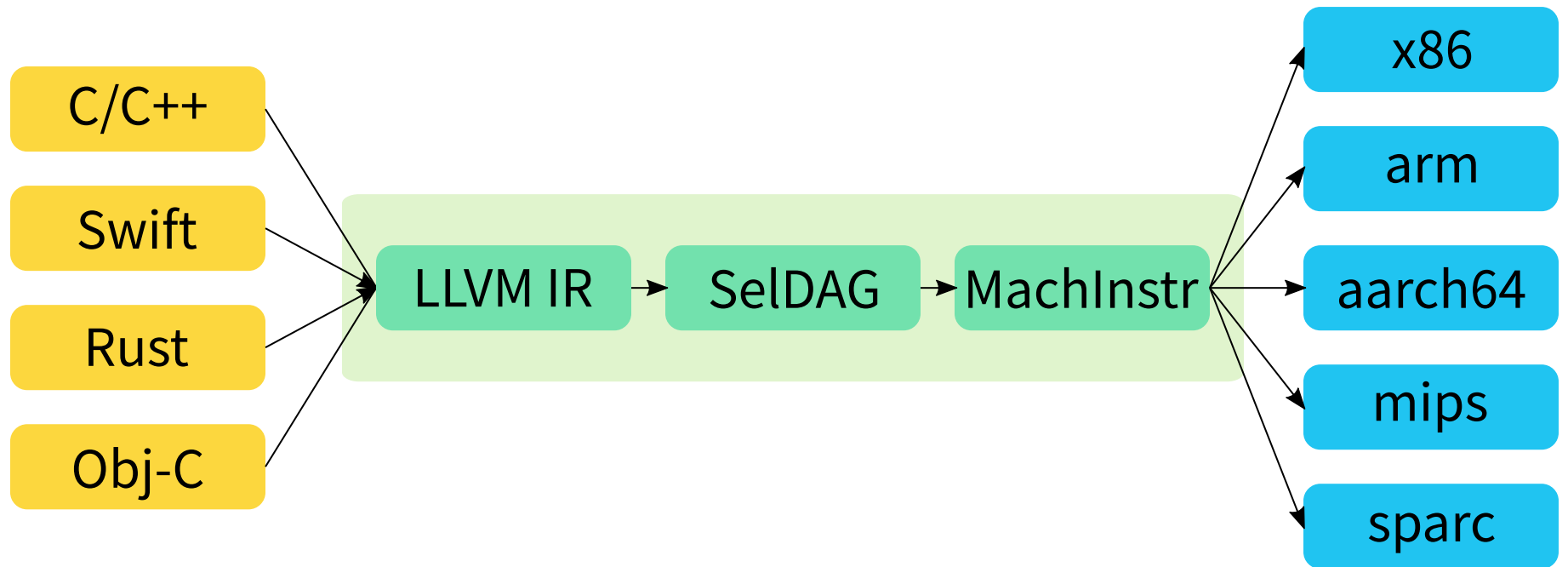
# INTERMEDIATE REPRESENTATION<sup>2/4</sup>



GCC Compiler Pipeline



# INTERMEDIATE REPRESENTATION<sup>3/4</sup>



LLVM Compiler Pipeline

# INTERMEDIATE REPRESENTATION<sup>4/4</sup>

GCC — GENERIC, GIMPLE, Tree SSA, and RTL.

LLVM — LLVM IR, Selection DAG, and Machine Instructions.

Java HotSpot — HIR, LIR, and MIR.

# CONTROL FLOW GRAPH<sup>1/3</sup>

**Basic block** — A sequence of instructions that will only be entered from the top and exited from the end.

**Edge** — If the basic block  $s$  may branch to  $t$ , then we add a directed edge  $(s, t)$ .

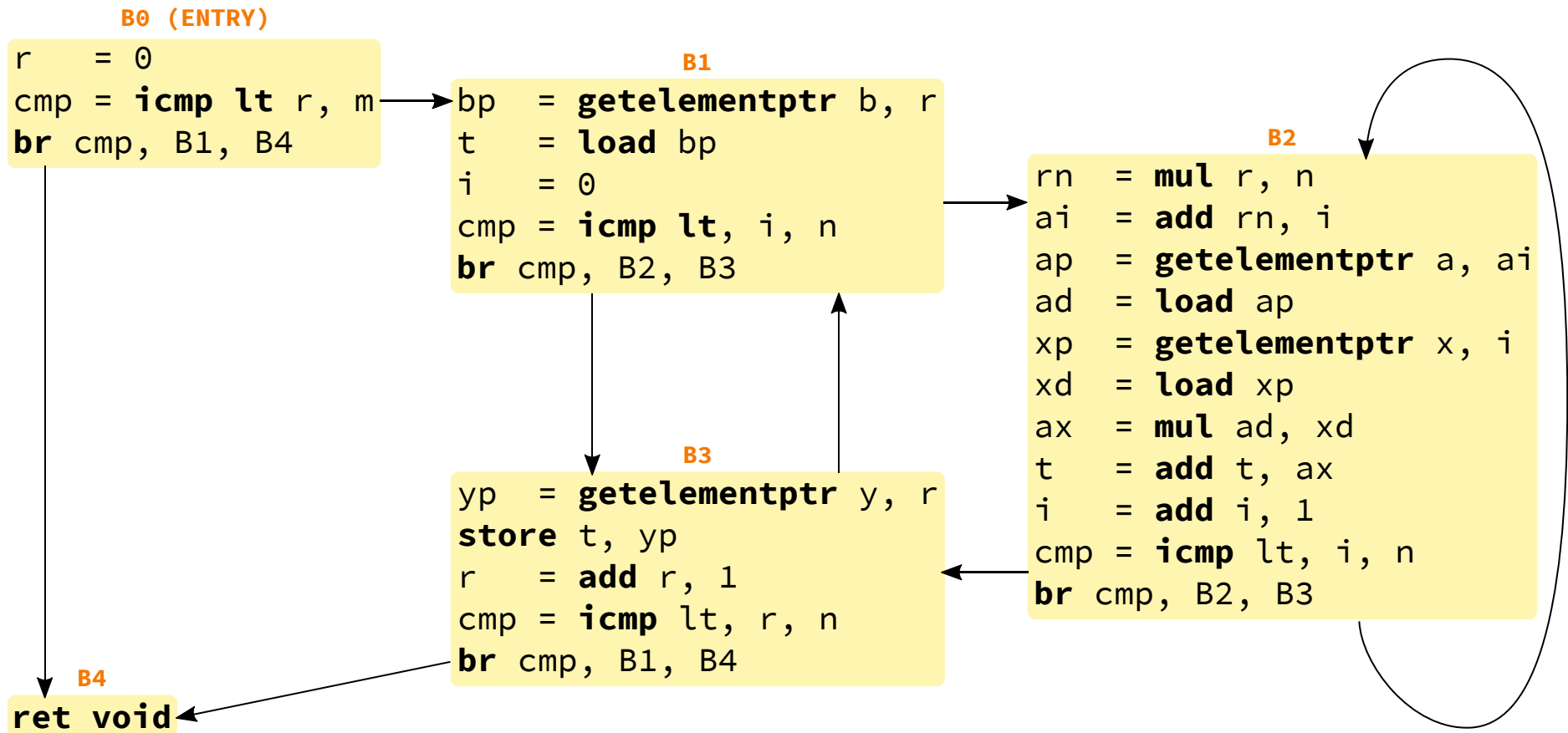
**Predecessor/Successor** — If there is an edge  $(s, t)$ , then  $s$  is a predecessor of  $t$  and  $t$  is a successor of  $s$ .

# CONTROL FLOW GRAPH<sup>2/3</sup>

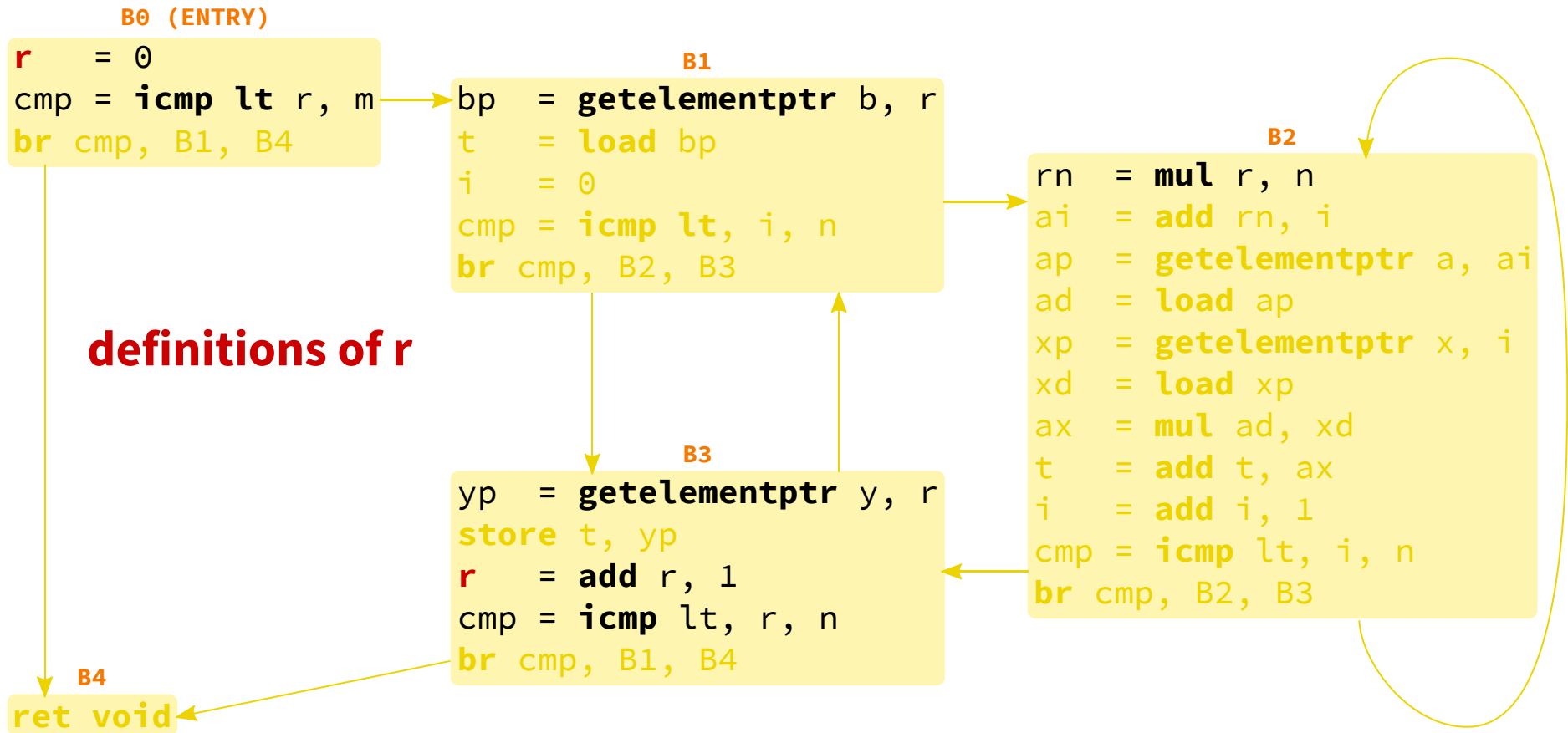
```
// y = a * x + b;
void matmul(double *restrict y,
            unsigned long m, unsigned long n,
            const double *restrict a,
            const double *restrict x,
            const double *restrict b) {
    for (unsigned long r = 0; r < m; ++r) {
        double t = b[r];
        for (unsigned long i = 0; i < n; ++i) {
            t += a[r * n + i] * x[i];
        }
        y[r] = t;
    }
}
```

Input source program

# CONTROL FLOW GRAPH<sup>3/3</sup>

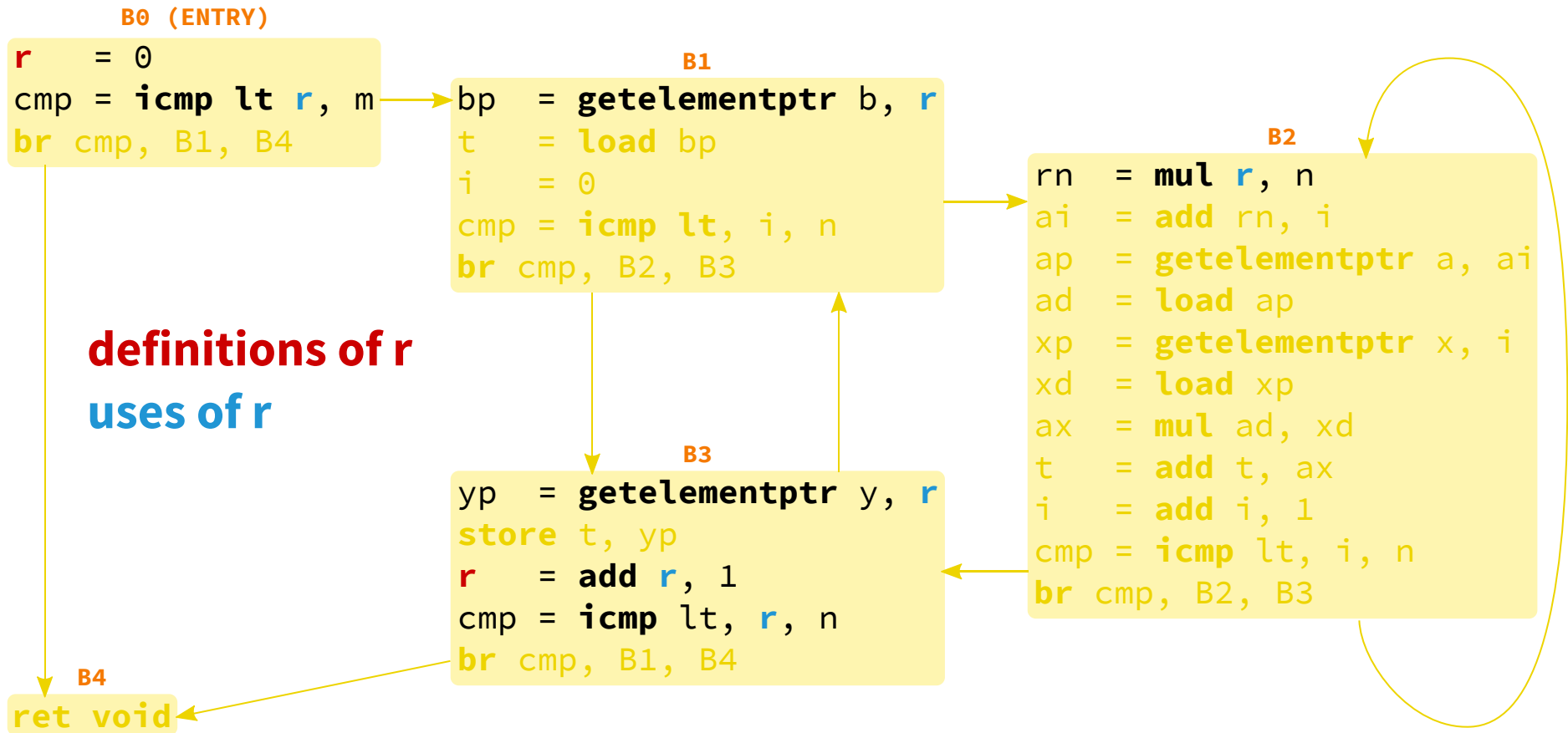


# VARIABLE DEFINITION



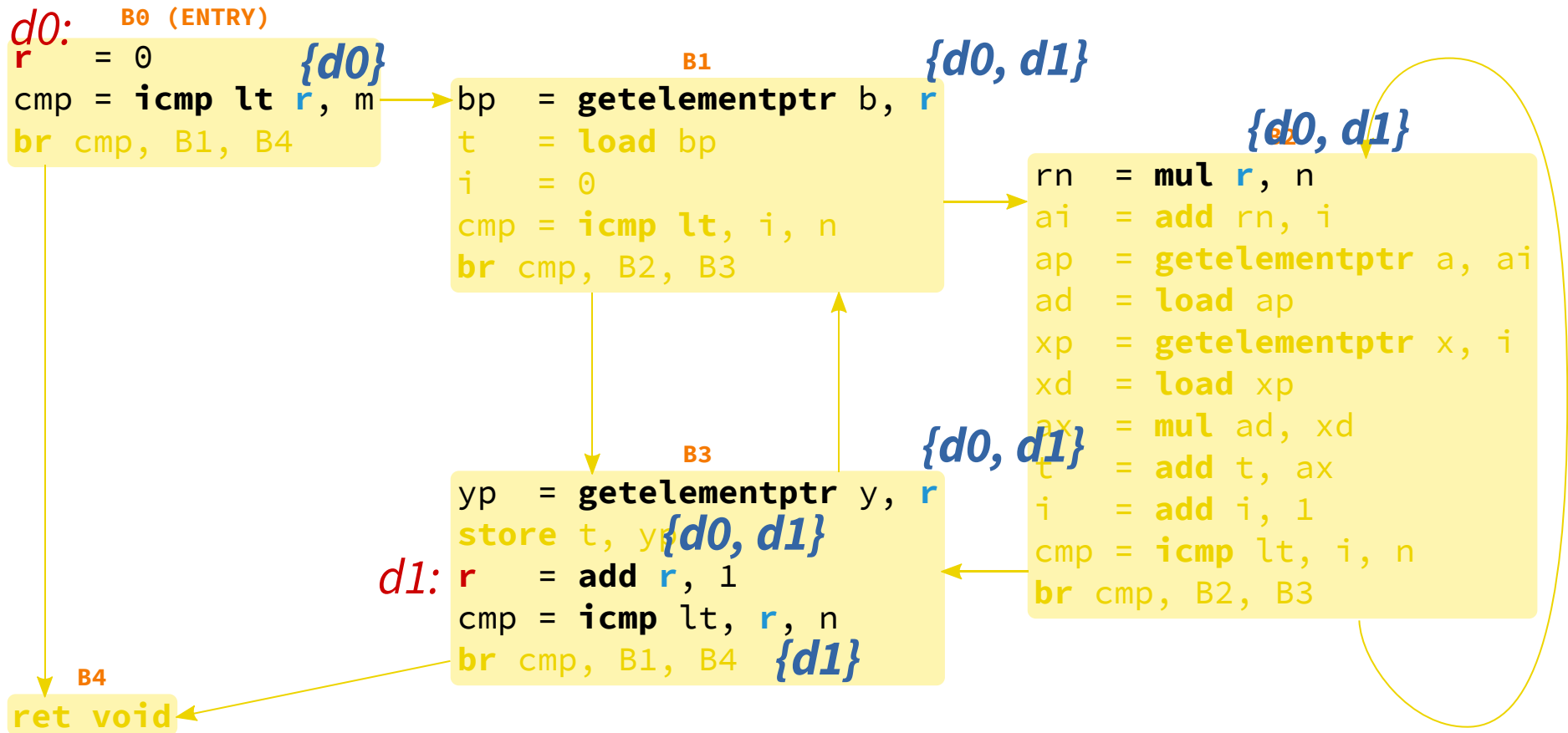
The place where a variable is **assigned** or **defined**.

# VARIABLE USE



The places where a variable is **referred** or **used**.

# REACHING DEFINITION<sup>1/2</sup>



Definitions that reaches a use.



# REACHING DEFINITION<sup>2/2</sup>

Constant propagation is a good example to show the usefulness of reaching definition.

```
void test(int cond) {
    int a = 1;    // d0
    int b = 2;    // d1
    if (cond) {
        c = 3;    // d2
    } else {
        // ReachDef[a] = {d0}
        // ReachDef[b] = {d1}
        c = a + b; // d3
    }

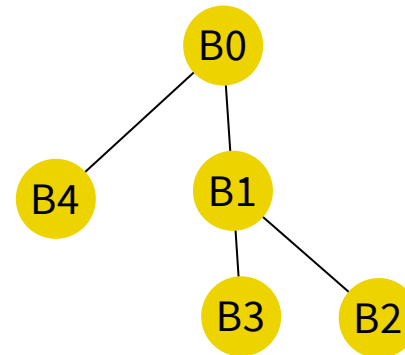
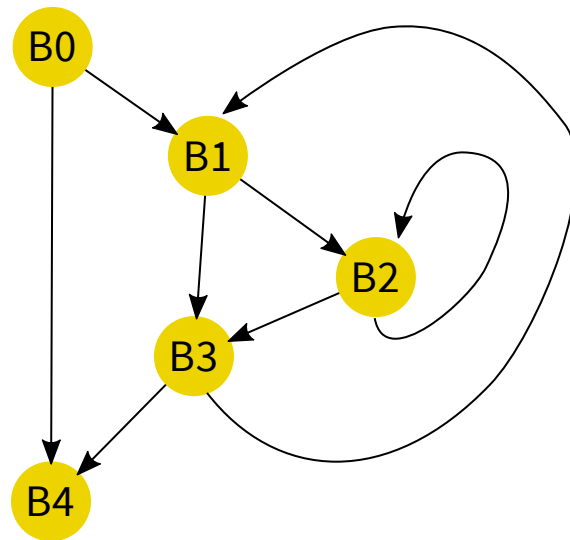
    // ReachDef[c] = {d2, d3}
    use(c);
}
```

# DOMINANCE RELATION<sup>1/2</sup>

A basic block  $s$  **dominates**  $t$  iff every path that goes from *entry* to  $t$  will pass through  $s$ .

Every basic block in a CFG has an immediate dominator and forms a *dominator tree*.

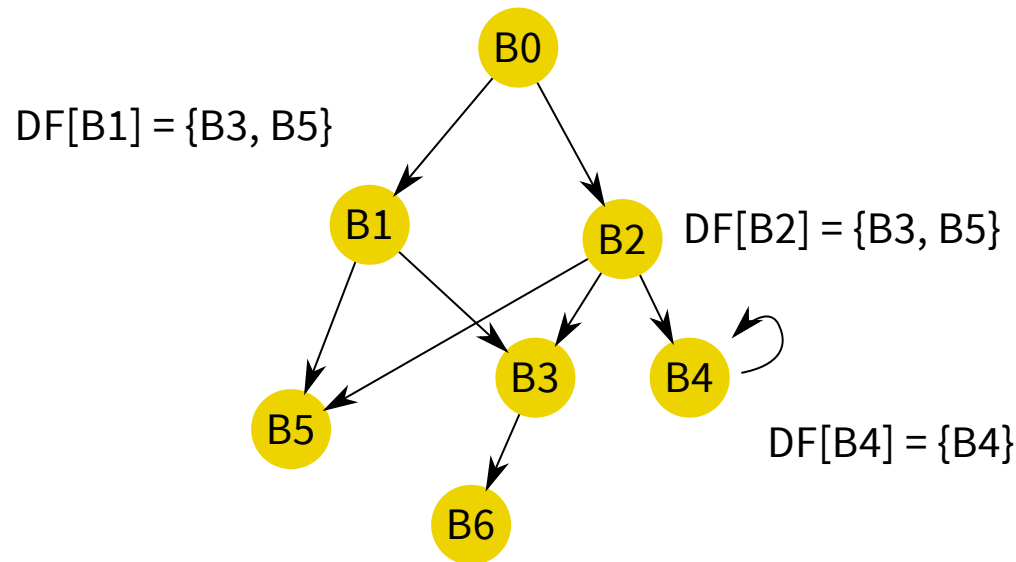
# DOMINANCE RELATION<sup>2/2</sup>



Dominator Tree

# DOMINANCE FRONTIER

A basic block  $t$  is a **dominance frontier** of a basic block  $s$ , if one of predecessor of  $t$  is dominated by  $s$  but  $t$  is not strictly dominated by  $s$ .



# STATIC SINGLE-ASSIGNMENT FORM

**static** — A static analysis to the program (not the execution.)

**single-assignment** — Every variable can only be assigned once.

SSA form is the most popular intermediate representation recently.

It is adopted by a wide range of compilers, such as GCC, LLVM, Java HotSpot, Android ART, etc.

# SSA PROPERTIES

Every variables can only be defined once.

Every uses can only refer to one definition.

Use  $\phi$  function to handle the merged control-flow.

# PHI FUNCTIONS

```
define void @foo(i1 cond,  
                  i32 a, i32 b) {  
ent:  
    br cond, b1, b2  
b1:  
    t0 = mul a, 4  
    br b3  
b2:  
    t1 = mul b, 5  
    br b3  
b3:  
    t2 = phi (t0), (t1)  
    use(t2)  
    ret  
}
```

```
define void @foo(i32 n) {  
ent:  
    br loop  
loop:  
    i0 = phi (0), (i1)  
    cmp = icmp ge i0, n  
    br cmp, end, cont  
cont:  
    use(i0)  
    i1 = add i0, 1  
    br loop  
end:  
    ret  
}
```

# ADVANTAGE OF SSA FORM

Compact — Reduce the **def-use chain**.

Referential transparency — The properties associated with a variable will not be changed, aka. context-free.



# REDUCED DEF-USE CHAIN

```
void foo(int cond1, int cond2,
         int a, int b) {
    int t;
    if (cond1) {
        t = a * 4; // d0
    } else {
        t = b * 5; // d1
    }
    if (cond2) {
        // reach-def: {d0, d1}
        use(t);
    } else {
        // reach-def: {d0, d1}
        use(t);
    }
}
```

```
void foo(int cond1, int cond2,
         int a, int b) {
    if (cond1) {
        t.0 = a * 4;
    } else {
        t.1 = b * 5;
    }
    t.2 = phi(t.0, t.1);
    if (cond2) {
        use(t.2);
    } else {
        use(t.2);
    }
}
```

# REFERENTIAL TRANSPARENCY<sup>1/2</sup>

```
void foo() {  
    int r = 5;  // d0  
  
    // ... SOME CODE ...  
  
    // We can only assume  
    // "r == 5" if d0 is the  
    // only reaching definition.  
    use(r);  
}
```

```
void foo() {  
    r.0 = 5;  
  
    // ... SOME CODE ...  
  
    // No matter what code are  
    // skipped above, it is safe  
    // to replace following r.0  
    // with 5.  
    use(r.0);  
}
```

# REFERENTIAL TRANSPARENCY<sup>2/2</sup>

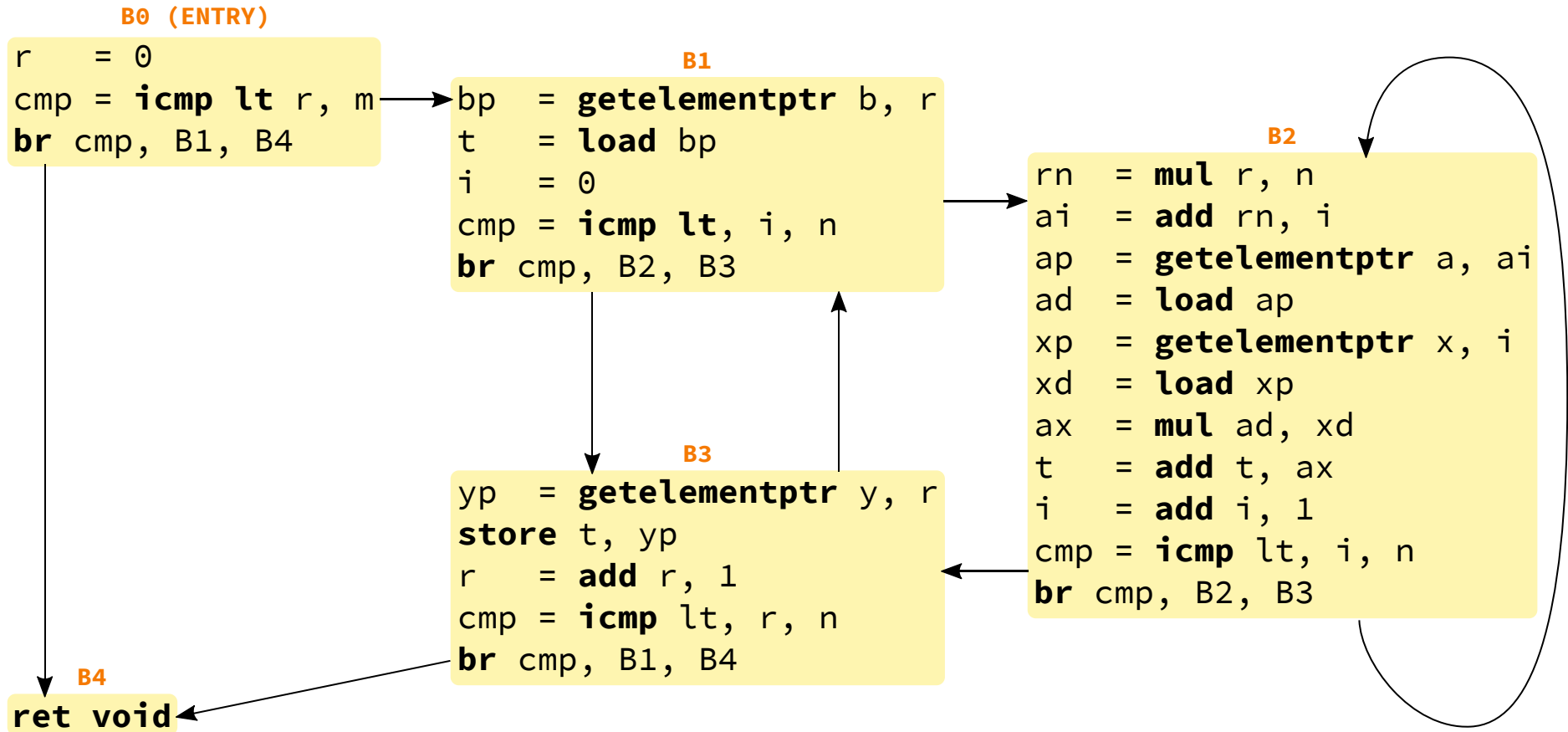
```
void foo(int a, int b) {  
    int c = a + b;  
    int r = a + b;  
    // Can we simply replace  
    // all occurrence of r with  
    // c? (NO)  
  
    // ... SOME CODE ...  
  
    use(r);  
}
```

```
void foo(int a, int b) {  
    c.0 = a + b;  
    r.0 = a + b;  
  
    // ... SOME CODE ...  
  
    // No matter what code are  
    // skipped above, it is safe  
    // to replace following r.0  
    // with c.0.  
    use(r.0);  
}
```

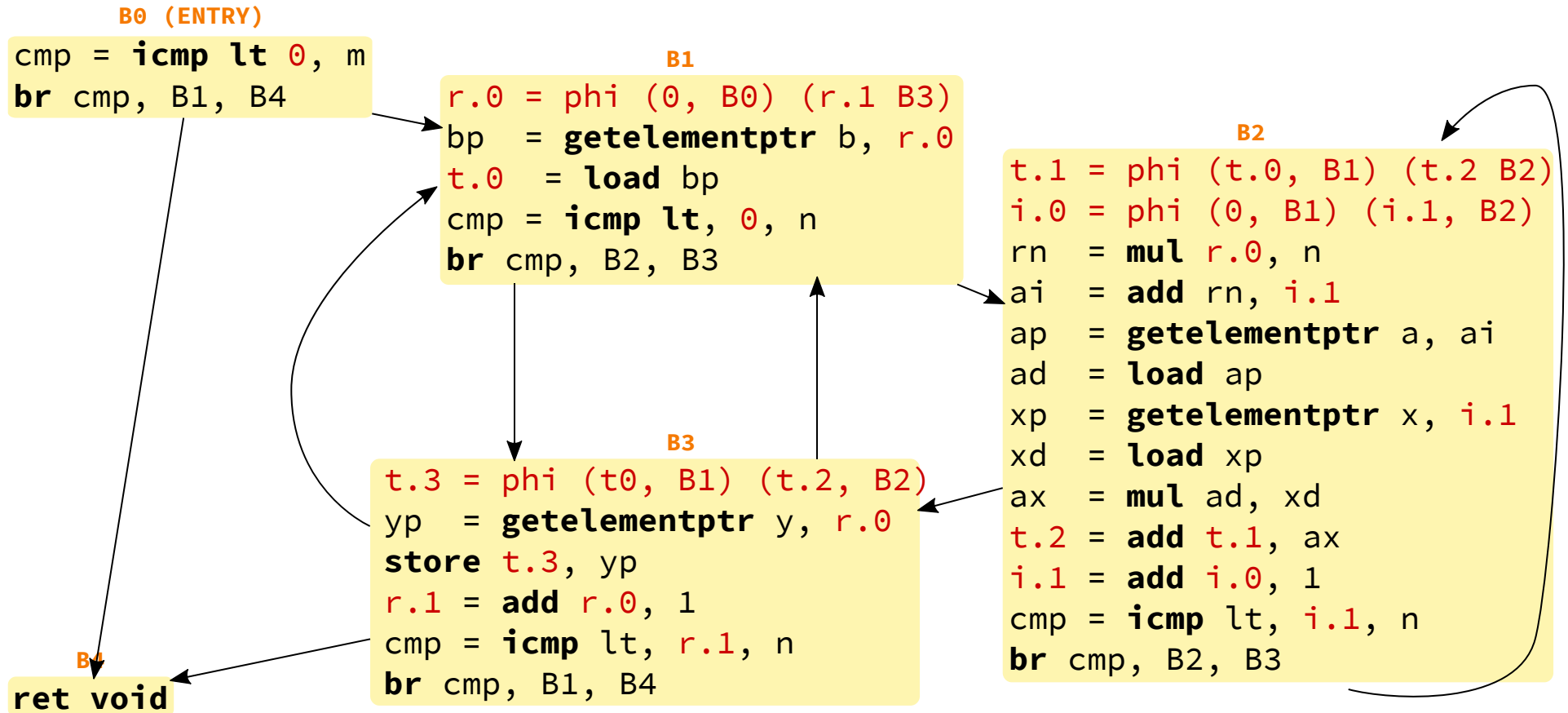
# BUILDING SSA FORM

1. Compute domination relationships between basic blocks and build the dominator tree.
2. Compute dominator frontiers.
3. Insert  $\phi$  functions at dominator frontiers.
4. Traverse the dominator tree and rename the variables.

# BEFORE SSA CONSTRUCTION



# AFTER SSA CONSTRUCTION



# OPTIMIZATIONS

# CONSTANT PROPAGATION<sup>1/3</sup>

Constant propagation, sometimes known as **constant folding**, will evaluate the instructions with constant operands and propagate the constant result.

```
a = add 2, 3  
b = a  
c = mul a, b
```

```
a = 5  
b = 5  
c = 25
```



# CONSTANT PROPAGATION<sup>2/3</sup>

Why do we need constant propagation?

```
struct queue *create_queue() {  
    return (struct queue*)malloc(sizeof(struct queue *) * 16);  
}
```

```
int process_data(int a, int b, int c) {  
    int k[] = { 0x13, 0x17, 0x19 };  
    if (DEBUG) {  
        verify_data(k, data);  
    }  
    return (a * k[1] * k[2] + b * k[0] * k[2] + c * k[0] * k[1]);  
}
```

# CONSTANT PROPAGATION<sup>3/3</sup>

For each basic block  $b$  from CFG in **reversed postorder**:

For each instruction  $i$  in basic block  $b$  from top to bottom:

If all of its operands are constants, the operation has no side-effect, and we know how to evaluate it at compile time, then evaluate the result, remove the instruction, and replace all usages with the result.

# GLOBAL VALUE NUMBERING<sup>1/2</sup>

Global value numbering tries to give numbers to the computed expression and maps the newly visited expression to the visited ones.

```
a = c * d;    // [(*, c, d) -> a]
e = c;        // [(*, c, d) -> a]
f = e * d;    // query: is (*, c, d) available?
use(a, e, f);
```

```
a = c * d;
use(a, c, a);
```

# GLOBAL VALUE NUMBERING<sup>2/2</sup>

Traverse the basic blocks in **depth-first** order on **dominator tree**.

Maintain a stack of hash table. Once we have returned from a child node on the dominator tree, then we have to pop the stack top.

Visit the instructions in the basic block with  $t = op\ a, b$  form and compute the hash for  $(op, a, b)$ .

If  $(op, a, b)$  is already in the hash table, then change

$t = op\ a, b$  with  $t = hash\_tab[(op, a, b)]$

Otherwise, insert  $(op, a, b) \rightarrow t$  to the hash table.

# DEAD CODE ELIMINATION<sup>1/6</sup>

Dead code elimination (DCE) removes **unreachable instructions** or **ignored results**.

Constant propagation might reveal more dead code since the branch conditions become constant value.

On the other hand, DCE can exploit more constant for constant propagation because several definitions are removed from the program.

# DEAD CODE ELIMINATION<sup>2/6</sup>

Conditional statements with constant condition

```
if (kIsDebugBuild) {           // Dead code
    check_invariant(a, b, c, d); // Dead code
}
```

# DEAD CODE ELIMINATION<sup>3/6</sup>

Platform-specific constant

```
void hash_ent_set_val(struct hash_ent *h, int v) {  
    if (sizeof(int) <= sizeof(void *)) {  
        h->p = (void *) (uintptr_t) (v);  
    } else {  
        h->p = malloc(sizeof(int));    // Dead code  
        *(h->p) = v;                  // Dead code  
    }  
}
```

# DEAD CODE ELIMINATION<sup>4/6</sup>

Computed result ignored

```
int compute_sum(int a, int b) {  
    int sum = (a + b) * (a - b + 1) / 2; // Dead code: Not used  
    return 0;  
}
```

Dead code after dead store optimization

```
void test(int *p, bool cond, int a, int b, int c) {  
    if (cond) {  
        t = a + b; // Dead code  
        *p = t;    // Will be removed by DSE  
    }  
    *p = c;  
}
```



# DEAD CODE ELIMINATION<sup>5/6</sup>

Dead code after code specialization

```
void matmul(double *restrict y, unsigned long m, unsigned long n,  
            const double *restrict a,  
            const double *restrict x,  
            const double *restrict b) {  
    if (n == 0) {  
        for (unsigned long r = 0; r < m; ++r) {  
            double t = b[r];  
            for (unsigned long i = 0; i < n; ++i) { // Dead code  
                t += a[r * n + i] * x[i];          // Dead code  
            }                                       // Dead code  
            y[r] = t;  
        }  
    } else {  
        // ... skipped ...  
    }  
}
```

# DEAD CODE ELIMINATION<sup>6/6</sup>

Traverse the CFG starting from *entry* in **reversed post order**.

Only traverse the successor that may be visited, i.e. if the branch condition is a constant, then ignore the other side.

While traversing the basic block, clear the dead instructions within the basic block.

After the traversal, remove the unvisited basic blocks and remove the variable uses that refers to the variables that are defined in the unvisited basic blocks.

# LOOP OPTIMIZATIONS

It is reasonable to assume a program spends more time in the loop body. Thus, loop optimization is an important issue in the compiler.

# LOOP INVARIANT CODE MOTION<sup>1/3</sup>

Loop invariant code motion (LICM) is an optimization which moves *loop invariants* or *loop constants* out of the loop.

```
int test(int n, int a, int b, int c) {  
    int sum = 0;  
    for (int i = 0; i < n; ++i) {  
        sum += i * a * b * c; // a*b*c is loop invariant  
    }  
    return sum;  
}
```

# LOOP INVARIANT CODE MOTION<sup>2/3</sup>

```
for (unsigned long r = 0; r < m; ++r) {  
    double t = b[r];  
    for (unsigned long i = 0; i < n; ++i) {  
        t += a[(r * n) + i] * x[i]; // "r*n" is a inner loop invariant  
    }  
    y[r] = t;  
}
```

```
for (unsigned long r = 0; r < m; ++r) {  
    double t = b[r];  
    unsigned long k = r * n; // "r*n" moved out of the inner loop  
    for (unsigned long i = 0; i < n; ++i) {  
        t += a[k + i] * x[i];  
    }  
    y[r] = t;  
}
```

# LOOP INVARIANT CODE MOTION<sup>3/3</sup>

How do we know whether a variable is a loop invariant?

If the computation of a value is not (transitively) depending on following black lists, we can assume such value is a loop invariant:

- **Phi instructions** associating with the loop being considered
- Instructions with **side-effects** or **non-pure** instructions, e.g. function calls

# INDUCTION VARIABLE

If **x** represents the trip count (iteration count), then we call  **$a*x+b$**  induction variables.

```
for (unsigned long r = 0; r < m; ++r) {  
    double t = b[r];  
    unsigned long k = r * n; // outer loop IV  
    for (unsigned long i = 0; i < n; ++i) {  
        unsigned long m = k + i; // inner loop IV  
        t += a[m] * x[i];  
    }  
    y[r] = t;  
}
```

**k** and **r** are induction variables of outer loop.

**i** and **m** are induction variables of inner loop.

# LOOP STRENGTH REDUCTION<sup>1/2</sup>

Strength reduction is an optimization to replace the multiplication in induction variables with an addition.

```
for (unsigned long r = 0; r < m; ++r) {  
    double t = b[r];  
    unsigned long k = r * n; // outer loop IV  
    for (unsigned long i = 0; i < n; ++i) {  
        unsigned long m = k + i; // inner loop IV  
        t += a[m] * x[i];  
    }  
    y[r] = t;  
}
```

```
for (unsigned long r = 0, k = 0; r < m; ++r, k += n) { // k  
    double t = b[r];  
    for (unsigned long i = 0, m = k; i < n; ++i, ++m) { // m  
        t += a[m] * x[i];  
    }  
    y[r] = t;  
}
```



# LOOP STRENGTH REDUCTION<sup>2/2</sup>

We can even rewrite the range to eliminate the index computation.

```
int sum(const int *a, int n) {  
    int res = 0;  
    for (int i = 0; i < n; ++i) {  
        res += a[i];    // implicit *(a + sizeof(int) * i)  
    }  
    return res;  
}
```

```
int sum(const int *a, int n) {  
    int res = 0;  
    const int *end = a + n;    // implicit a + sizeof(int) * n  
    for (const int *p = a; p != end; ++p) {    // range updated  
        res += *p;  
    }  
    return res;  
}
```

# LOOP UNROLLING<sup>1/2</sup>

Unroll the loop body multiple times.

Purpose: Reduce amortized loop iteration overhead.

Purpose: Reduce load/store stall and exploit instruction-level parallelism, e.g. software pipelining.

Purpose: Prepare for vectorization, e.g. SIMD.

# LOOP UNROLLING<sup>2/2</sup>

```
for (unsigned long r = 0, k = 0; r < m; ++r, k += n) {  
    double t = b[r];  
    switch (n & 0x3) { // Duff's device  
        case 3: t += a[k + 2] * x[2];  
        case 2: t += a[k + 1] * x[1];  
        case 1: t += a[k] * x[0];  
    }  
    for (unsigned long i = n & 0x3; i < n; i += 4) {  
        t += a[k + i] * x[i];  
        t += a[k + i + 1] * x[i + 1];  
        t += a[k + i + 2] * x[i + 2];  
        t += a[k + i + 3] * x[i + 3];  
    }  
    y[r] = t;  
}
```

# INSTRUCTION SELECTION<sup>1/5</sup>

Instruction selection is a process to map IR into **machine instructions**.

Compiler back-ends will perform **pattern matching** to the best select instructions (according to the heuristic.)

# INSTRUCTION SELECTION<sup>2/5</sup>

Complex operations, e.g. *shift-and-add* or *multiply-accumulate*.

Array load/store instructions, which can be translated to one *shift-add-and-load* on some architectures.

IR instructions are which *not natively supported* by the target machine.

# INSTRUCTION SELECTION<sup>3/5</sup>

```
define i64 @mac(i64 %a, i64 %b, i64 %c) {  
ent:  
    %0 = mul i64 %a, %b  
    %1 = add i64 %0, %c  
    ret i64 %1  
}
```

```
mac:  
    madd    x0, x1, x0, x2  
    ret
```

# INSTRUCTION SELECTION<sup>4/5</sup>

```
define i64 @load_shift(i64* %a, i64 %i) {  
ent:  
    %0 = getelementptr i64, i64* %a, i64 %i  
    %1 = load i64, i64* %0  
    ret i64 %1  
}
```

```
load_shift:  
    ldr    x0, [x0, x1, lsl #3]  
    ret
```

# INSTRUCTION SELECTION<sup>5/5</sup>

```
%struct.A = type { i64*, [16 x i64] }

define i64 @get_val(%struct.A* %p, i64 %i, i64 %j) {
ent:
    %0 = getelementptr %struct.A, %struct.A* %p, i64 %i, i32 1, i64 %j
    %1 = load i64, i64* %0
    ret i64 %1
}
```

```
get_val:
    movz    w8, #0x88
    madd    x8, x1, x8, x0
    add     x8, x8, x2, lsl #3
    ldr     x0, [x8, #8]
    ret
```



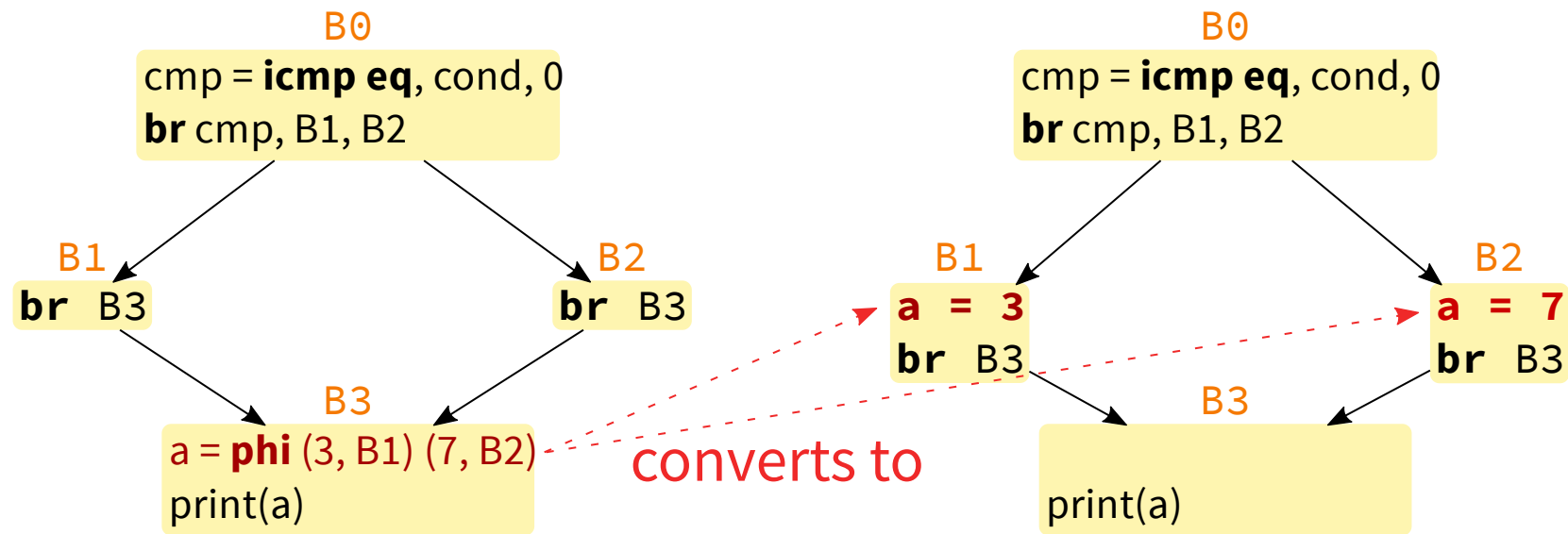
# SSA DESTRUCTION<sup>1/5</sup>

How do we deal with the **phi** functions?

Copy the assignment operator to the end of the predecessor.

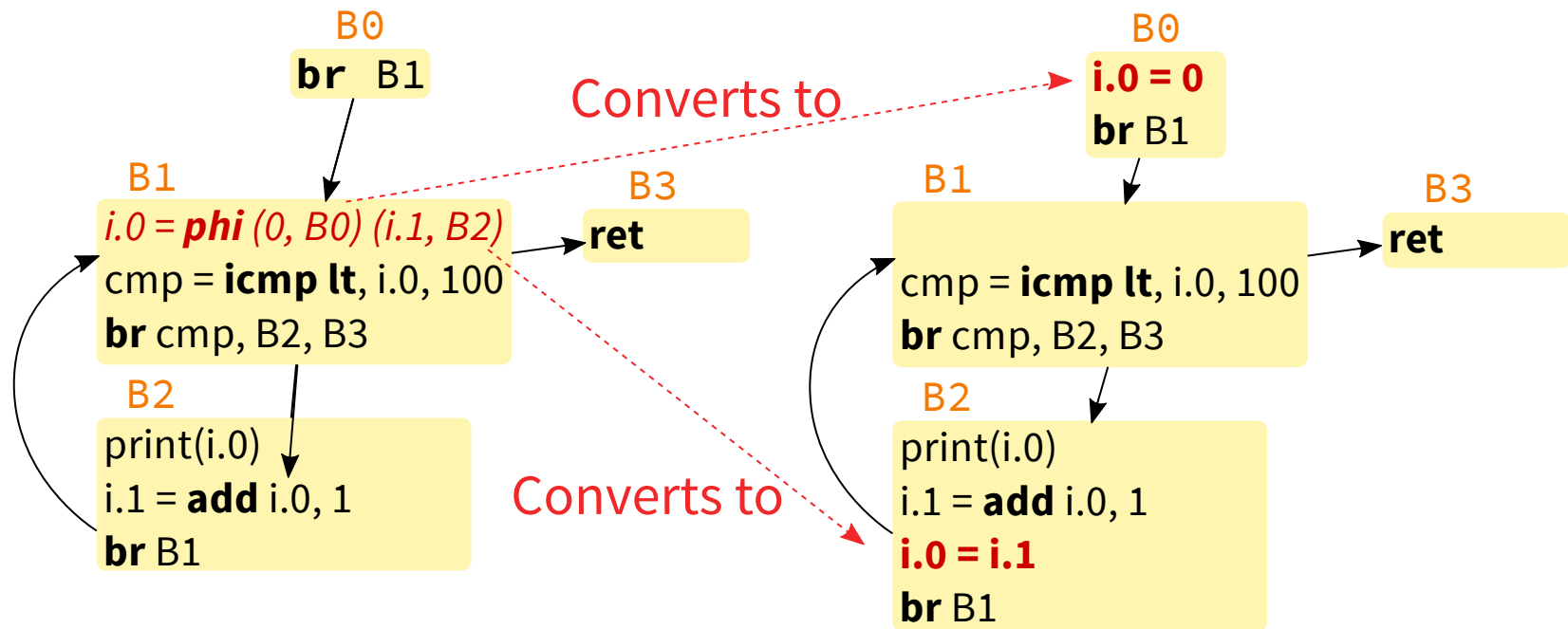
(Must be done carefully)

# SSA DESTRUCTION<sup>2/3</sup>



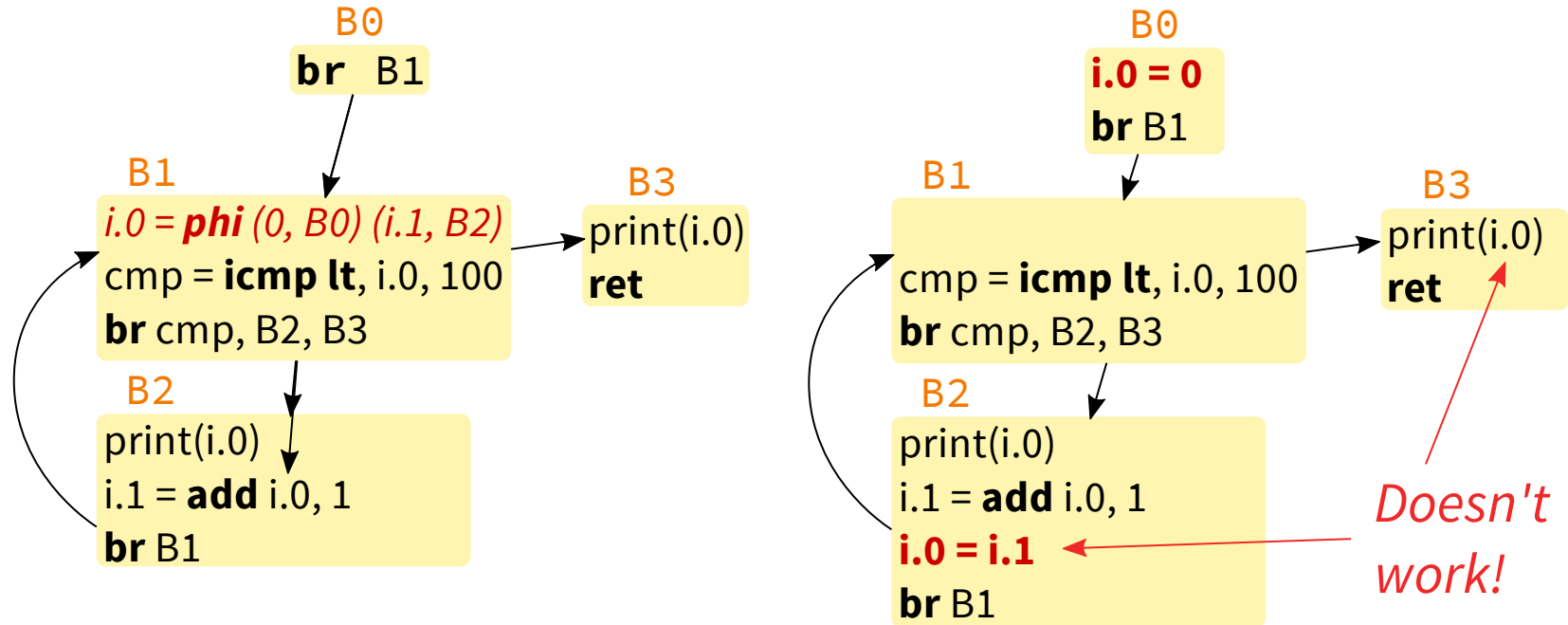
Example to show the assignment copying

# SSA DESTRUCTION<sup>3/5</sup>



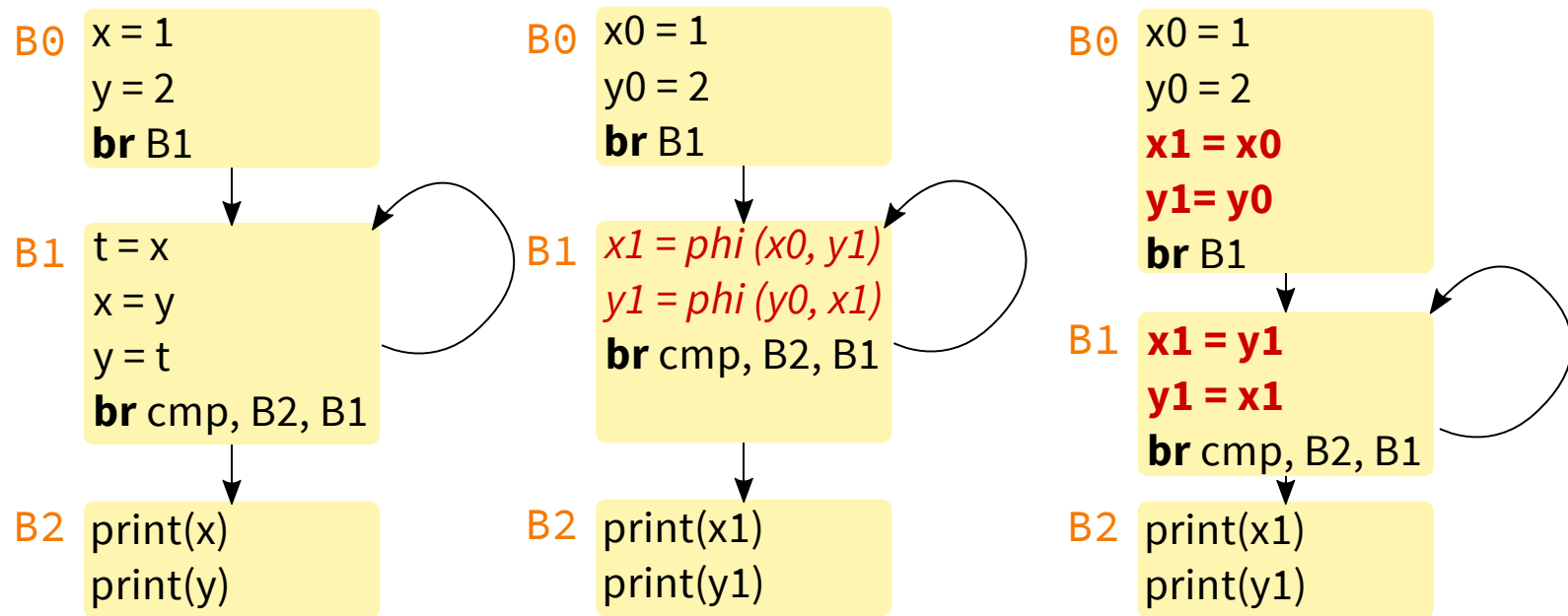
Example to show the assignment copying with loop

# SSA DESTRUCTION<sup>4/5</sup>



**Lost copy problem** — Naive de-SSA algorithm doesn't work due to live range conflicts. Need extra assignments and renaming to avoid conflicts.

# SSA DESTRUCTION<sup>5/5</sup>



**Swap problem** — Conflicts due to parallel assignment semantics of phi instructions. A correct algorithm should detect the cycle and implement parallel assignment with swap instructions.

# REGISTER ALLOCATION<sup>1/2</sup>

We have to replace infinite virtual registers with finite machine registers.

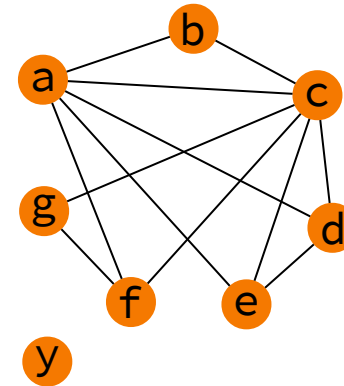
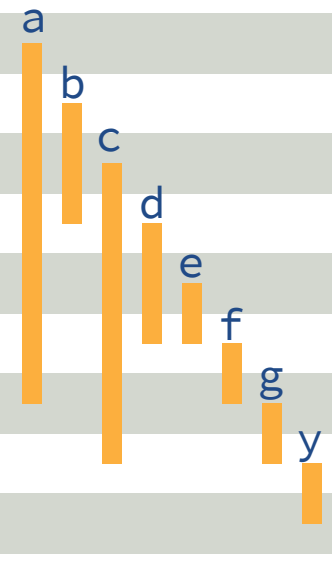
**Register Allocation** — Make sure that the maximum simultaneous register usages are less than  $k$ .

**Register Assignment** — Assign a register given the fact that registers are always sufficient.

The classical solution is to compute the **life time** of each variables, build the inference graph, spill the variable to stack if the inference graph is not  *$k$ -colorable*.

# REGISTER ALLOCATION<sup>2/2</sup>

```
mov a, #1
mov b, #1
mov c, #2
add d, a, b
add e, a, c
add f, d, e
add g, a, f
add y, c, g
ret y
```



# INSTRUCTION SCHEDULING<sup>1/2</sup>

Sort the instruction according the number of cycles in order to reduce execution time of the critical path.

Constraints: (a) Data dependence, (b) Functional units, (c) Issue width, (d) Datapath forwarding, (e) Instruction cycle time, etc.



# INSTRUCTION SCHEDULING<sup>2/2</sup>

```
0: add r0, r0, r1
1: add r1, r2, r2
2: ldr r4, [r3]
3: sub r4, r4, r0
4: str r4, [r1]
```

```
2: ldr r4, [r3] # load instruction needs more cycles
0: add r0, r0, r1
1: add r1, r2, r2
3: sub r4, r4, r0
4: str r4, [r1]
```

# COMPILER AND ICT INDUSTRY

**WHERE CAN WE FIND COMPILER?**

# SMART PHONES

- Android ART (Java VM)
- OpenGL|ES GLSL Compiler (Graphics)



# BROWSERS

- Javascript-related
  - Javascript Engine, e.g. V8, IonMonkey, JavaScriptCore, Chakra
  - Regular Expression Engine
  - WebAssembly
- WebGL — OpenGL binding for the Web (Graphics)
- WebCL — OpenCL binding for the Web (Computing)



# DESKTOP APPLICATION RUN-TIME ENVIRONMENTS

- Java run-time (Java, Scala, Clojure, Groovy)
- .NET run-time (C#, F#)
- Ruby
- Python
- PHP

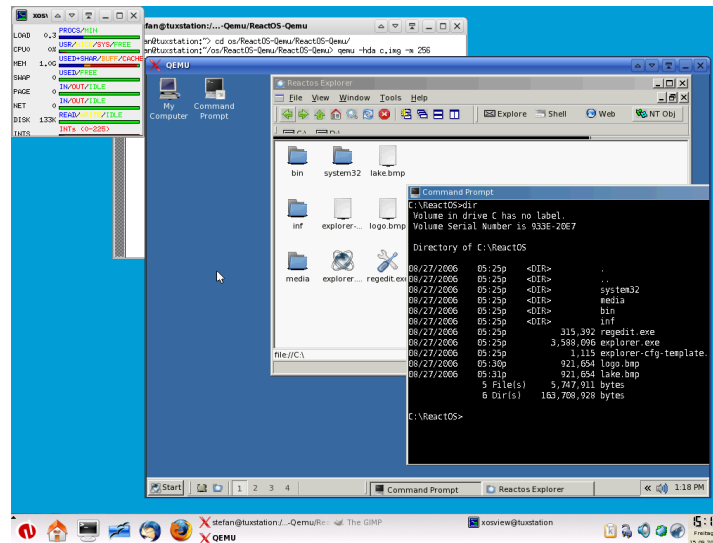


python™



# VIRTUAL MACHINES AND EMULATORS

- Simulate different architecture (using dynamic recompilation), e.g. QEMU
- Simulate special instructions that are not supported by hypervisor.



# DEVELOPMENT ENVIRONMENT

- C/C++ compiler, e.g. MSVC, GCC, LLVM
- Java compiler, e.g. OpenJDK
- DSP compilers for digital signal processors.
- VHDL compilers for IC design team.
- Profiling and/or instrumentation tools, e.g. Valgrind, Dr. Memory, Address Sanitizer, Memory Sanitizer



# WHAT DOES A COMPILER TEAM DO?

Develop the compiler toolchain for the in-house processors,  
e.g. DSP, GPU, CPU, and etc.

Tune the performance of the existing compiler or virtual  
machines.

Develop a new programming language or model (e.g. Cuda  
from NVIDIA.)

# SOME RESEARCH TOPICS ...

**Memory model for concurrency** — Designing a good memory model at programming language level, which should be intuitive to human while open for compiler/architecture optimization, is still a challenging problem.

Both Java and C++ took efforts to standardize. However, there are still some unintuitive cases that are allowed and some desirable cases being ruled out.

# THE END

Q & A