Phase 5 Overview

6.1120

Fall 2025

Recitation 8 overview

- Deliverables (deadlines, objectives, etc.)
- Optimization overview

Deliverables (Due December 8 @ 11:59 pm ET)

- An optimized version of your VM from Phase 3
  - Can implement as many optimizations as you'd like, but must implement at least one of the following:
    - Threaded code + stack caching
    - Tagged pointers
    - Type specialization
    - Function inlining
    - Any other optimization that requires dataflow analysis
    - Machine code generation
    - Shape analysis
  - **Credit will only be given if at least one of the optimizations above is implemented (and implementation must be general)**

Deliverables (Due December 8 @ 11:59 pm ET)

- An optimized version of your VM from Phase 4
  - Must support `--mem N` as before
  - Must also support the following additional options:
    - `--opt all` to enable all optimizations
    - `--opt <optname>` to enable `optname` optimization (if the optimization affects code generation, such as inlining)
    - `--emit-code` to print out any code that your VM generates
      - Will not be used for automated testing, just nice for manual inspection

Deliverables (Due December 8 @ 11:59 pm ET)

- Writeup
  - **Must clearly identify all optimizations that are implemented and specify where in code base each optimization is implemented**
  - Describe each optimization using diagrams and code examples
  - Should convincingly argue how each optimization is beneficial, general, and correct
  - Should discuss other optimizations that were considered but not implemented

Milestone (Due November 14 @ 10 pm ET)

- Consists of preliminary writeup
- Should describe what optimizations you plan on implementing (with justifications for why you believe they will be beneficial)
- Should discuss other optimizations that were considered

Extra Credit Checkpoint (Due November 24 @ 10 pm ET)

- Score based on the “base grade” of your compiler at the time of submission
- Grading based on correctness and derby performance (relative to baseline)
- Maximum of 5% EC on P5 grade

Grading

- 50% based on implementation (checked for correctness)
  - 25% based on public benchmarks
  - 25% based on hidden tests
- 30% based on derby performance
- 20% based on report
  - But will also be used to help grade your implementation

P5: Performance Optimization

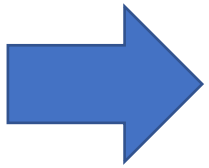
https://grow.acorns.com/reasons-it-can-pay-to-use-cash/

Performance Engineering

- “Premature optimization is the root of all evil”
- Always benchmark!
  - Make sure you measure run-times rigorously
  - Take minimum for serial execution
- Optimization stages
  - High-level design
  - Algorithmic improvements
  - Memory allocation
  - Cache efficiency
  - Low-level improvements
    - Compiler intrinsics
    - Bit hacks
    - Branching improvement

(Subroutine-)Threaded Code

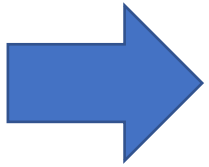
```
inst = {LoadLocal, LoadLocal,
  Add, StoreLocal, ...};

for (; inst != end; ++inst) {
  switch (inst->op) {
    case LoadLocal:
      load_local(...); break;
    case StoreLocal:
      store_local(...); break;
    case Add:
      add(...); break;
  }
}
```



```
load_local(...);
load_local(...);
add(...);
store_local(...)
;
```

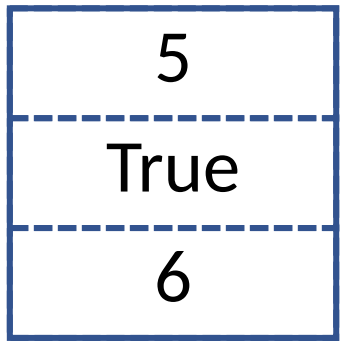
(Direct-)Threaded Code

```
inst = {LoadLocal, LoadLocal,
  Add, StoreLocal, ...};

for (; inst != end; ++inst) {
  switch (inst->op) {
    case LoadLocal:
      load_local(...); break;
    case StoreLocal:
      store_local(...); break;
    case Add:
      add(...); break;
  }
}
```

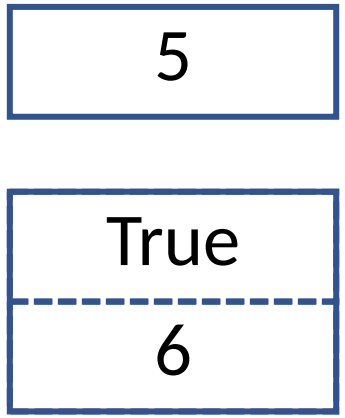


```
inst = {&&load_local,
  &&load_local, &&add,
  &&store_local, ...};

goto **inst++;
load_local:
  // load local var.
  goto **inst++;
store_local:
  // store local var.
  goto **inst++;
add:
  // perform addition
  goto **inst++;
```

Stack Caching

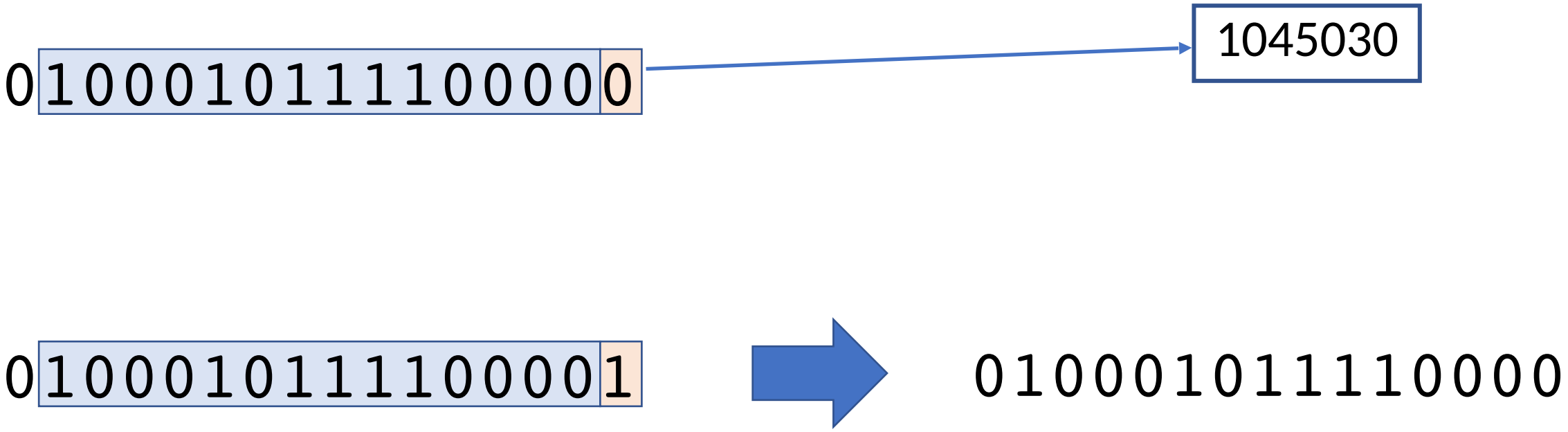



```
std:stack<Value*> operands;
...
case Neg:
  Value* op = operands.pop();
  Value* neg = new Integer(-(op->getInt()));
  operands.push(neg);
```




```
Value* top; std:stack<Value*> rest;
...
case Neg:
  Value* op = top;
  Value* neg = new Integer(-(op->getInt()));
  top = neg;
```

Tagged Pointers



Type Specialization (Cast Elimination)

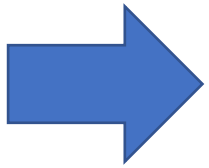
```
inst = {LoadConst, Neg,...};

for (; inst != end; ++inst) {
  switch (inst->op) {
    case Neg:
      if (operand->type() != Int) {
        throw InvalidCastException();
      }
      ret = new Integer(-(operand->getInt()));
      ...
  }
}
```

Type Specialization (Cast Elimination)

```
inst = {LoadIntConst, NegFast,...};

for (; inst != end; ++inst) {
  switch (inst->op) {
    case NegFast:
      ret = new Integer(-(operand->getInt()));
      ...
  }
}
```

Type Specialization (Unboxing)

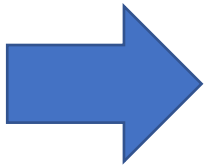
```
Value* a = new Integer(1);
Value* b = new Integer(2);
int aval = a->getInt();
int bval = b->getInt();
int cval = aval + bval;
Value* c = new
Integer(cval);
```



```
int a = 1;
int b = 2;
int cval = a + b;
Value* c = new
Integer(cval);
```

Function Inlining

```
double = fun(x) {
  return 2 * x;
};

add = fun(x, y) {
  return x + y;
};

while (i < 100) {
  i = double(i);
  i = add(i, 1);
}
```

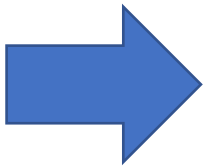


```
while (i < 100)
{
  i = 2 * i;
  i = i + 1;
}
```

Dataflow Optimizations

Copy propagation:

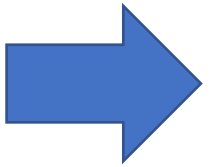
```
a = b;
c = a * 2;
d = e;
f = d;
g = f / h;
```



```
c = b * 2;
g = e / h;
```

Dead code elimination:

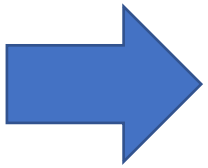
```
a = (3 > 2);
if (a) {
  r = r / 2;
} else {
  r = r + 1;
}
```



```
r = r / 2;
```

Machine Code Generation

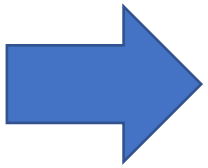
```
function {
  local_vars = [y, x],
  constants = [None,
2],
  instructions = [
    load_local 0
    load_const 1
    sub
    store_local 1
    load_const 0
    return
  ]
}
```



```
__execute_0 :
   push %rdi
   call assert_integer
   pop %rax
   shr $3 %rax
   mov $2 %rcx
   sub %rcx %rax
   ret
```

Machine Code Generation

```
function {
  local_vars = [y, x],
  constants = [None,
2],
  instructions = [
    load_local 0
    load_const 1
    sub
    store_local 1
    load_const 0
    return
  ]
}
```

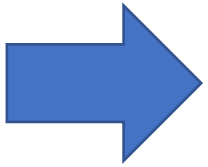


```
__execute_0 :
   push %rdi
   call assert_integer
   pop %rax
   shr $3 %rax
   mov $2 %rcx
   sub %rcx %rax
   ret
```

Remove tag

Machine Code Generation

```
function {
  local_vars = [y, x],
  constants = [None,
2],
  instructions = [
    load_local 0
    load_const 1
    sub
    store_local 1
    load_const 0
    return
  ]
}
```



```
__execute_0 :
   push %rdi
   call assert_integer
   pop %rax
   shr $3 %rax
   mov $2 %rcx
   sub %rcx %rax
   ret
```

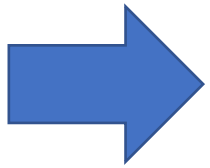
Remove tag

Perform subtract

Peephole Optimizations

Strength reduction:

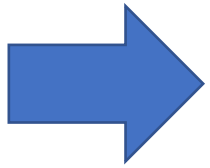
```
a = x / 8;
```



```
a = a >> 3;
```

Null sequence elimination:

```
load_const 0
store_local 0
load_local 0
```



```
load_const 0
```

Other Optimizations

- Redesign or replace MITScript VM bytecode
  - Add new instructions, replace with register-based IR, etc.
- Replace mark-and-sweep garbage collector
  - e.g., copying collector, generational collector, etc.
- Shape analysis of records (to be discussed in lecture)
- Custom data types for storage
  - Instead of using STL objects
  - Std::string and std::vector
  - Look into SwissTables (Google's optimized hash tables)