Lecture Week 9, Wednesday

Prof. Dr. Luc Bläser

Last Lecture - Quiz

How could we realize a GC for C/C++?

GC Needs to Detect Pointers

- Exact method by metadata
  - Pointer offsets in object by type descriptor
  - Descriptor for root set

Impossible for C/C++

- Conservative method by speculation
  - Check each word whether it could be a pointer
  - If yes, consider it as pointer

Possible for C/C++

Why does the first approach not work for C/C++?

No GC Metadata in C/C++

- Pointer = explicit numeric value
  - Can be converted to number type and backwards
- Every word could be a pointer
  - Independent of the type

```
long x = (long)ptr;
```

Conservative Method

- Boehm–Demers–Weiser GC for C/C++
- Plausibility test for each word in program whether it could point to an occupied object in heap
- If yes, consider it as pointer

What is the problem of this method?

Conservative Method

- More pointers may be collected through speculation than actually used

Possible memory leaks

- No so bad for Mark & Sweep GC
- Impossible for Compacting GC

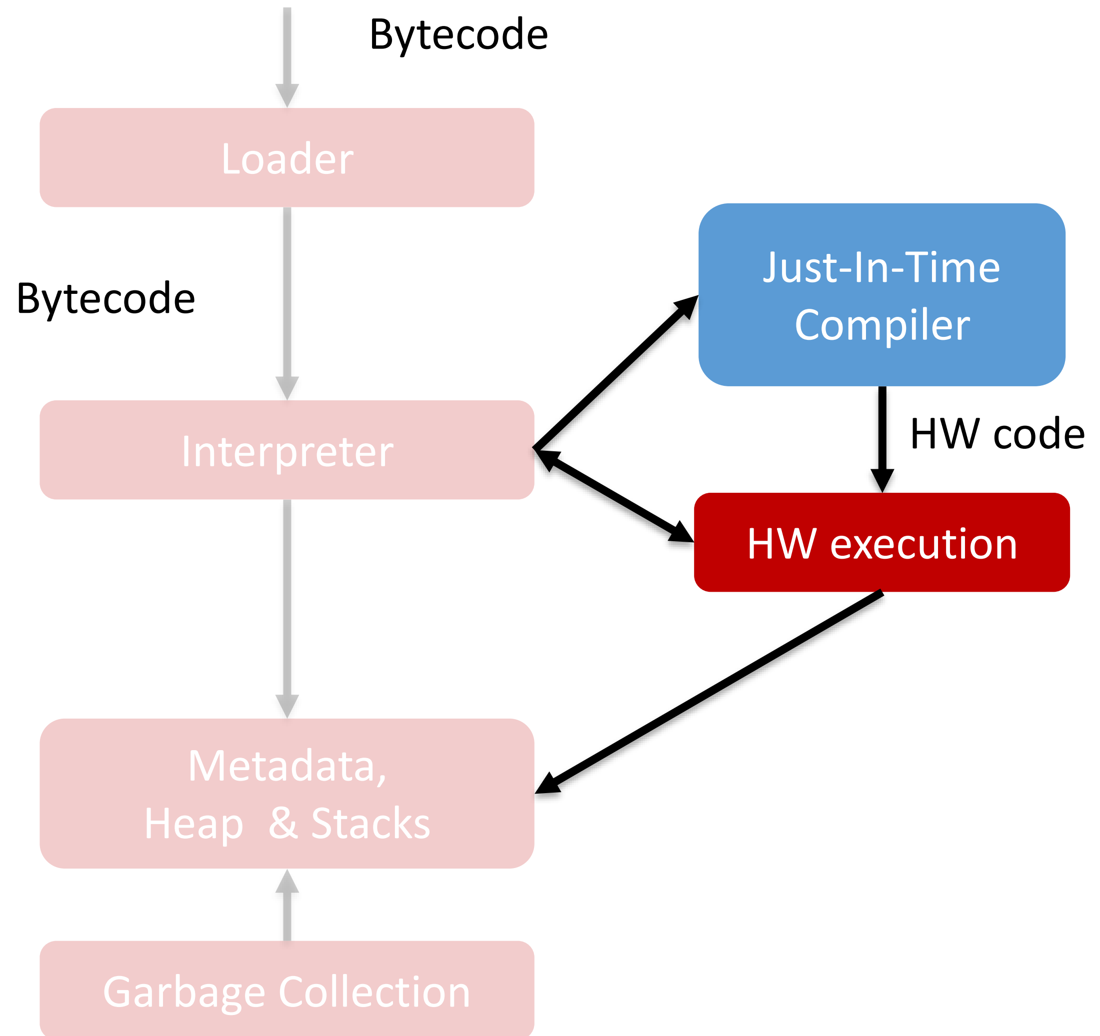
Today's Topics

- Just-In-Time compiler
- Intel 64 instruction set
- Register allocation

Learning Goals

- Understand the purpose and functionality of a Just-In-Time compiler
- Have an overview of the essential part of the Intel 64 instruction set
- Know how to implement a first part of the JIT-compiler for your virtual machine

Our Focus Today



Just-In-Time (JIT) Compiler

- Substantially faster execution than interpretation
- Translate bytecode to native HW code and run it
- Not necessarily everything, only critical pieces

Hot Spot

- Performance-critical code section
- Frequently executed, e.g. >= 50 times
- JIT compilation is then particularly beneficial

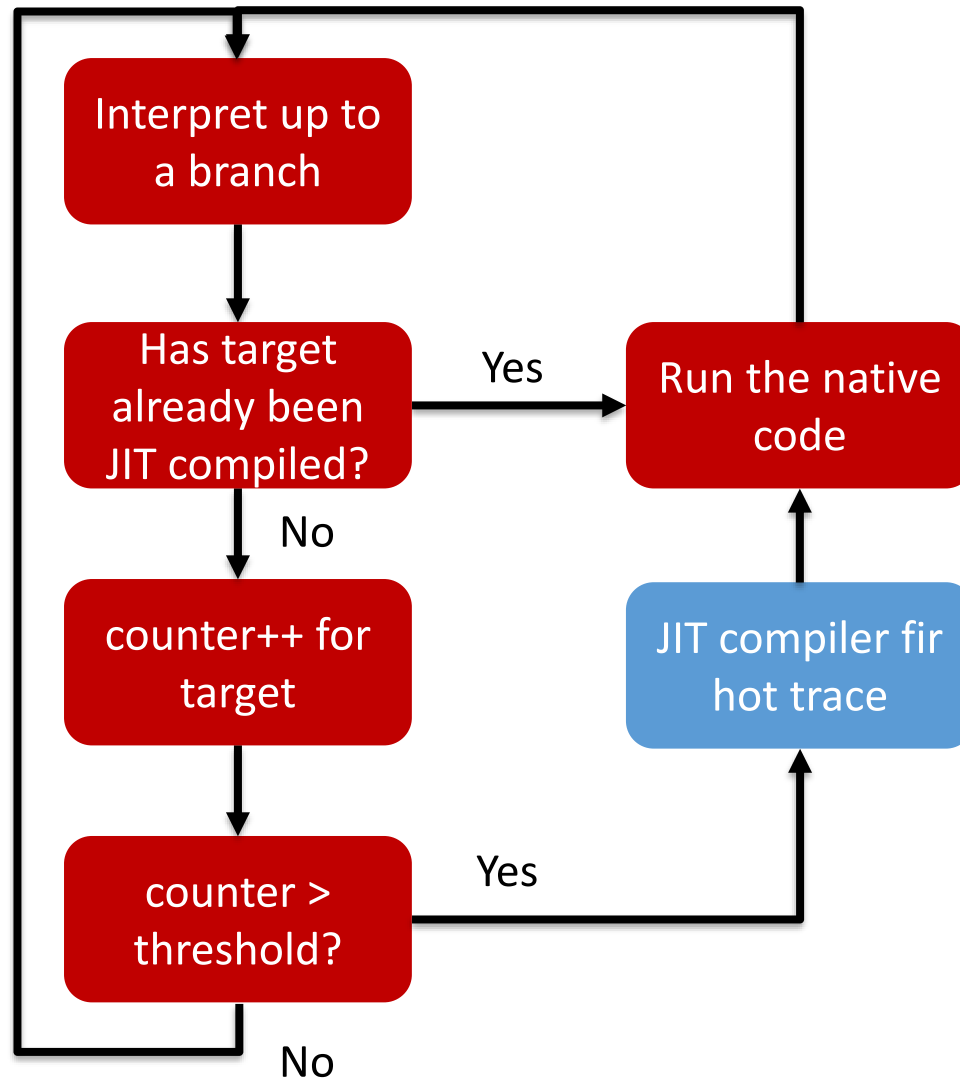
```
        ...
begin:  load 1
        load 2
        icmplt
        if_false end
        load 1
        ldc 1
        iadd
        store 1
        br begin
end:    ...
```

Hot spot (loop)

Profiling

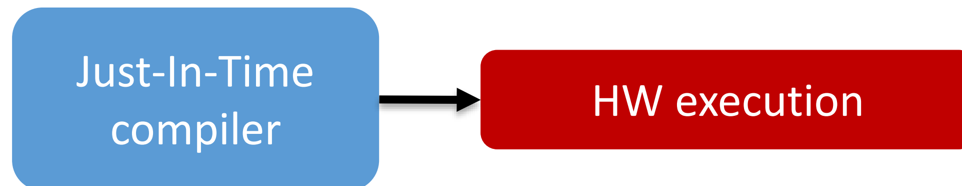
- Interpreter counts execution of certain code pieces
  - Methods
  - Traces (code paths)
- If frequently executed, initiate JIT for that piece

Approach



Native Execution

- Our target platform: Intel 64 architecture



- Register allocation
- Code generator
- Code linker

- Executable page
- Native stack
- Return to interpreter

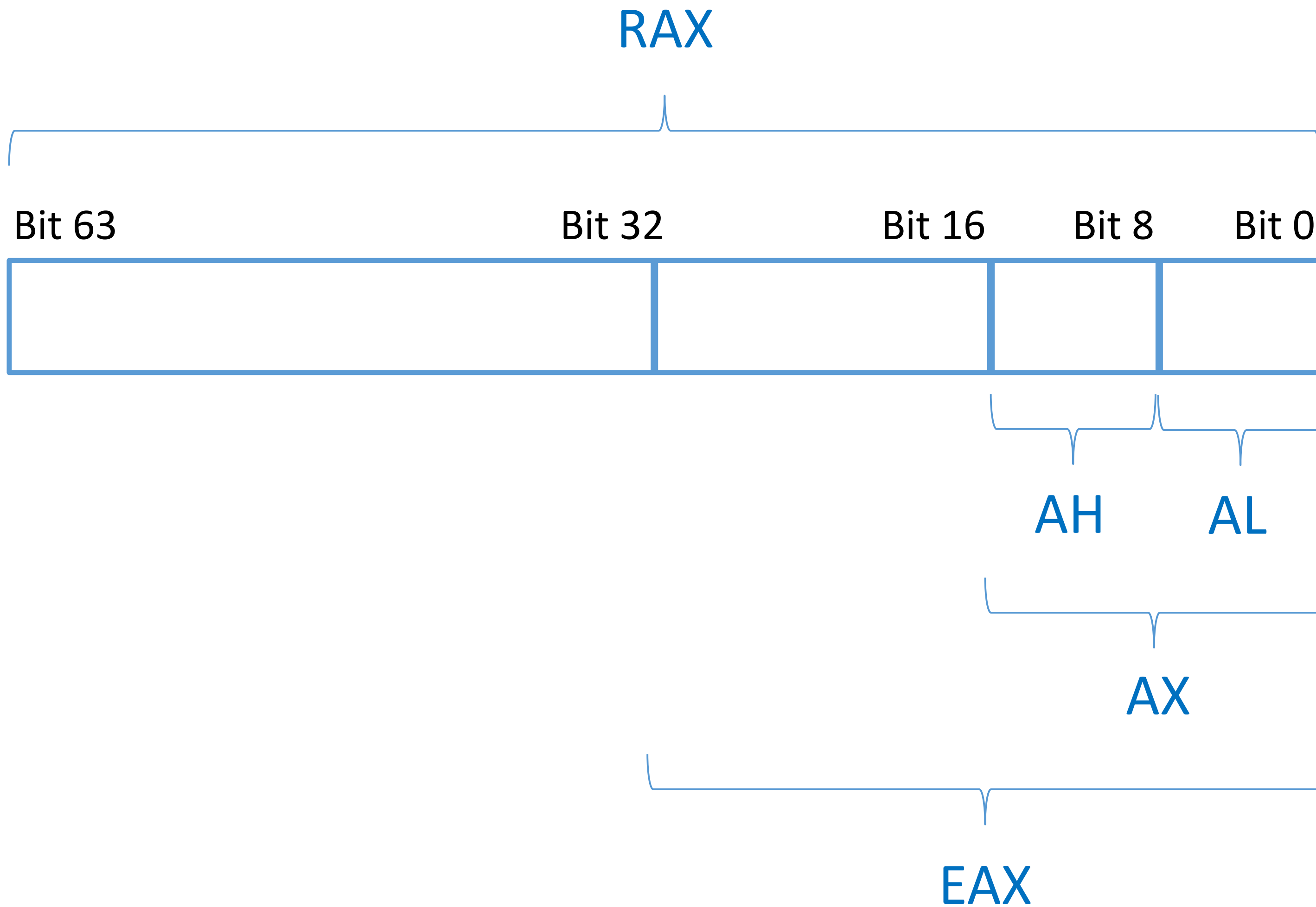
Intel 64 Architecture

- Instructions use registers (in contrast to bytecode)
- 14 general-purpose registers for 64-bit integers

RAX
RBX
RCX
RDX
RSI
RDI
R8
…
R15

Historical naming

Partial Register Accesses



Additional Intel 64 Registers

- Special registers (64-bit)

RSP	Stack Pointer
RBP	Base Pointer
RIP	Instruction Pointer

- Moreover, media register (64- and 128-bit)
- Floating points in FPU stack machine

Basic Instructions

- Move values from and to registers

`MOV RAX, 100`	Assign 100 to register RAX
`MOV RAX, RBX`	Copy RBX into RAX

Arithmetic Instructions

`ADD RAX, RBX`	RAX += RBX
`SUB RAX, RBX`	RAX -= RBX
`IMUL RAX, RBX`	RAX *= RBX
`IDIV RBX`	RDX must be 0 for non-negative RAX beforehand, otherwise -1. RAX /= RBX, RDX = RAX % RBX

Signed Mul and Div

IDIV uses fix registers (register clobbering)

Prepare IDIV

- IDIV is fix 128-bit division of RDX:RAX
- Set RDX depending on RAX
  - 0 if RAX >= 0
  - -1 if RAX < 0
- Use special instruction CDQ

CDQ	Signed convert RAX to RDX:RAX (convert to quad word)

Example

```
(x - 1) / 2
```

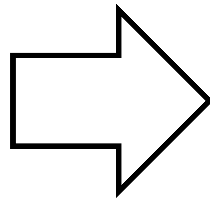
```
// assume x is stored in RAX
MOV RBX, 1
SUB RAX, RBX              // RAX = x - 1
MOV RBX, 2
CDQ                       // prepare RDX before IDIV
IDIV RBX                  // RAX = (x - 1) / 2
// result in RAX
```

Cross Compilation

Bytecode

```
load 1
ldc 1
isub
ldc 2
idiv
```

Stack machine



x64 Code

```
MOV RAX, …
MOV RBX, 1
SUB RAX, RBX
MOV RBX, 2
CDQ
IDIV RBX
```

Register machine

Translation Patterns

- x64 code fragment per bytecode instruction pattern

ldc <int_value>

```
MOV <reg0>, <int_value>
```

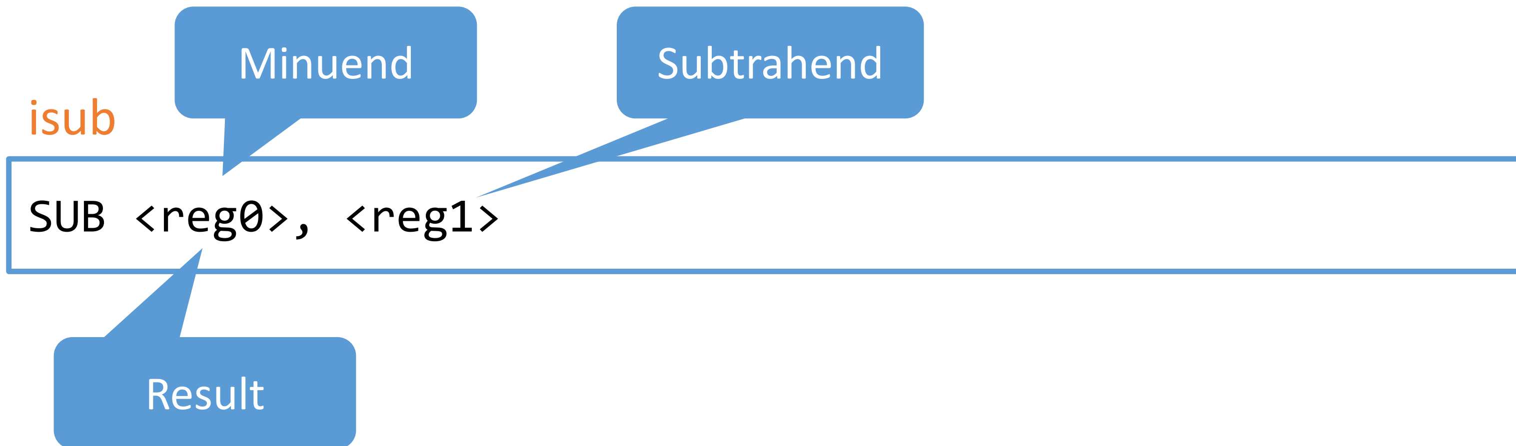
Register placeholder <reg0>

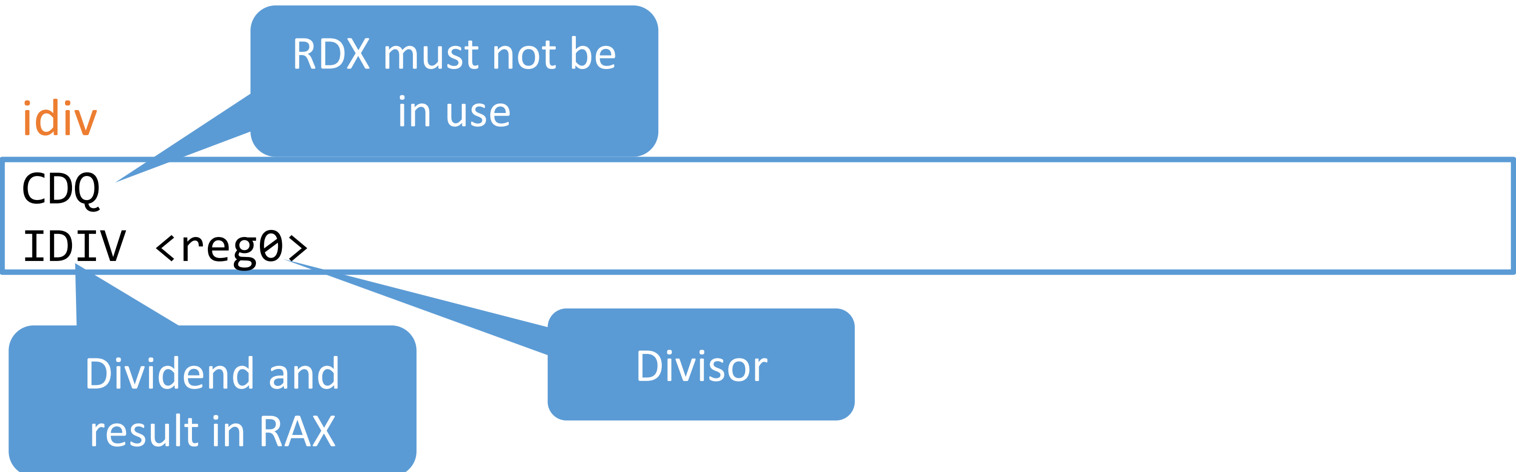
ineg

```
NEG <reg0>
```

Operand & result

Translation Patterns

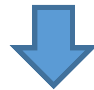




Register Allocation

- Determine correct registers for code templates
  - Registers to choose depends on preceding instructions
  - Instruction can make registers free or occupied

minuend in RAX
subtrahend in RBX



```
SUB <reg0>, <reg1>
```

<reg0> = RAX, <reg1> = RBX

RBX is free
result in RAX

Local vs. Global Register Allocation

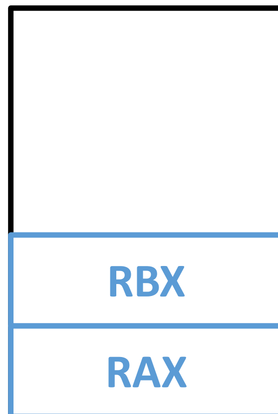
- Local register allocation
  - For expression evaluation (evaluation stack)
  - Map evaluation stack entry to register => register stack
- Global register allocation
  - Use registers for variables (in certain sections)
  - Much faster than memory accesses

However, we only have a limited number of registers

Local Register Allocation

- Cross-compiler uses a stack of occupied registers
  - Register stack corresponds to the evaluation stack that the interpreter would have used at that instruction
- Update the stack per translated bytecode instruction

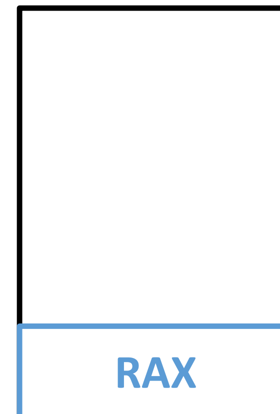
Register stack before



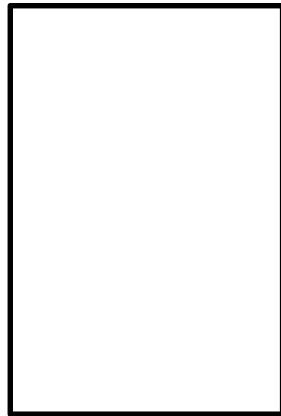
isub

SUB RAX, RBX

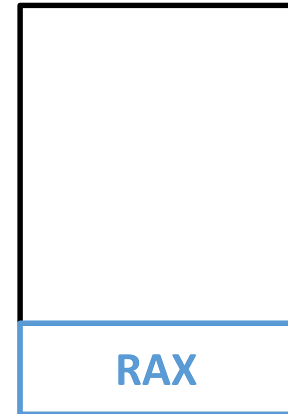
Register stack after



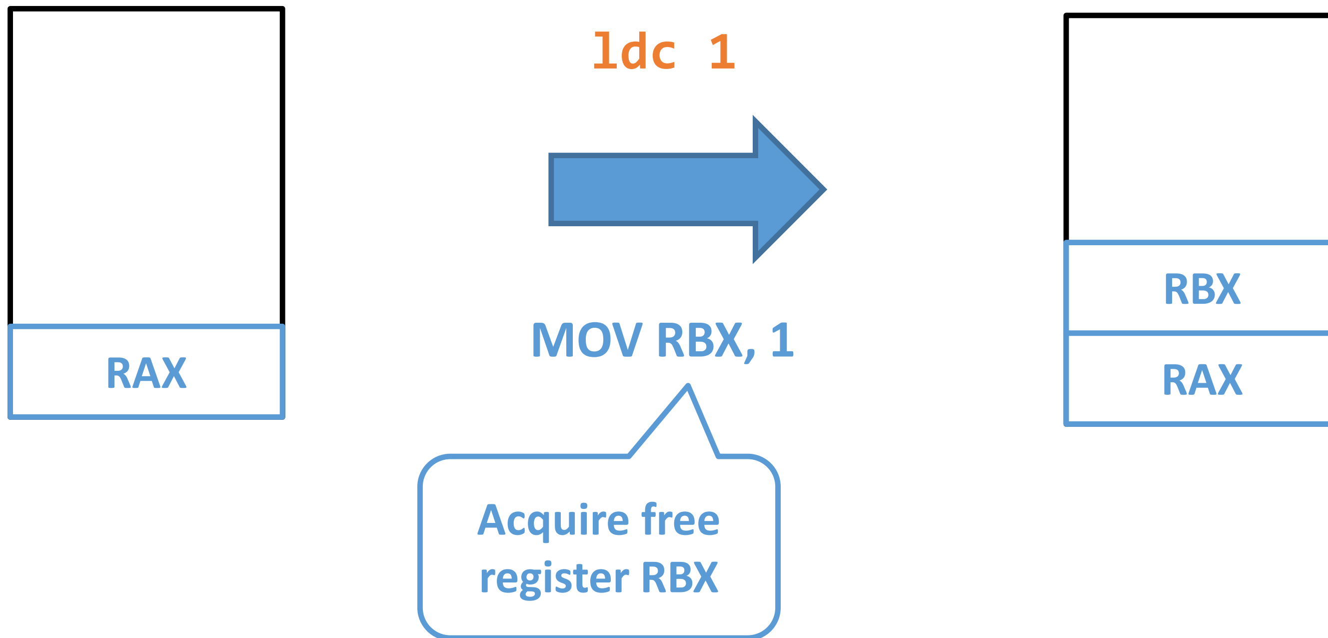
Translation


empty

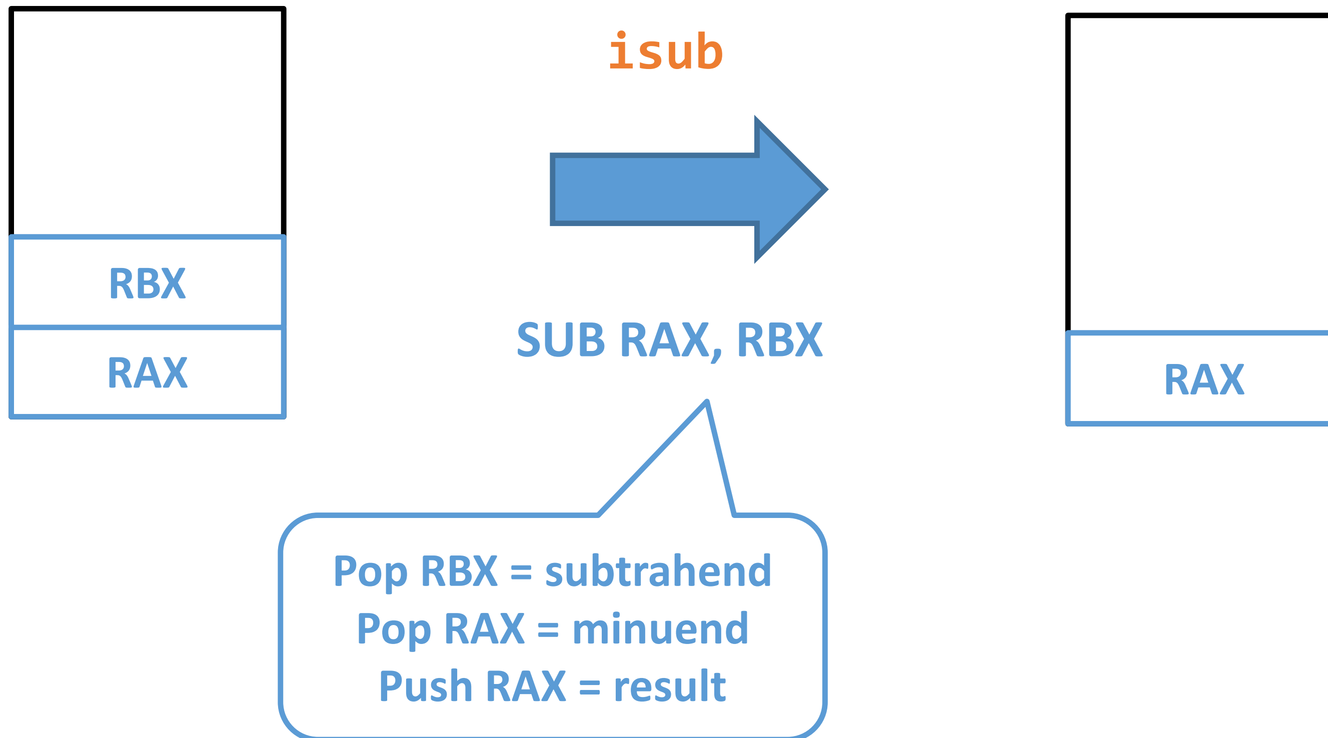
We assume param x is already in RAX



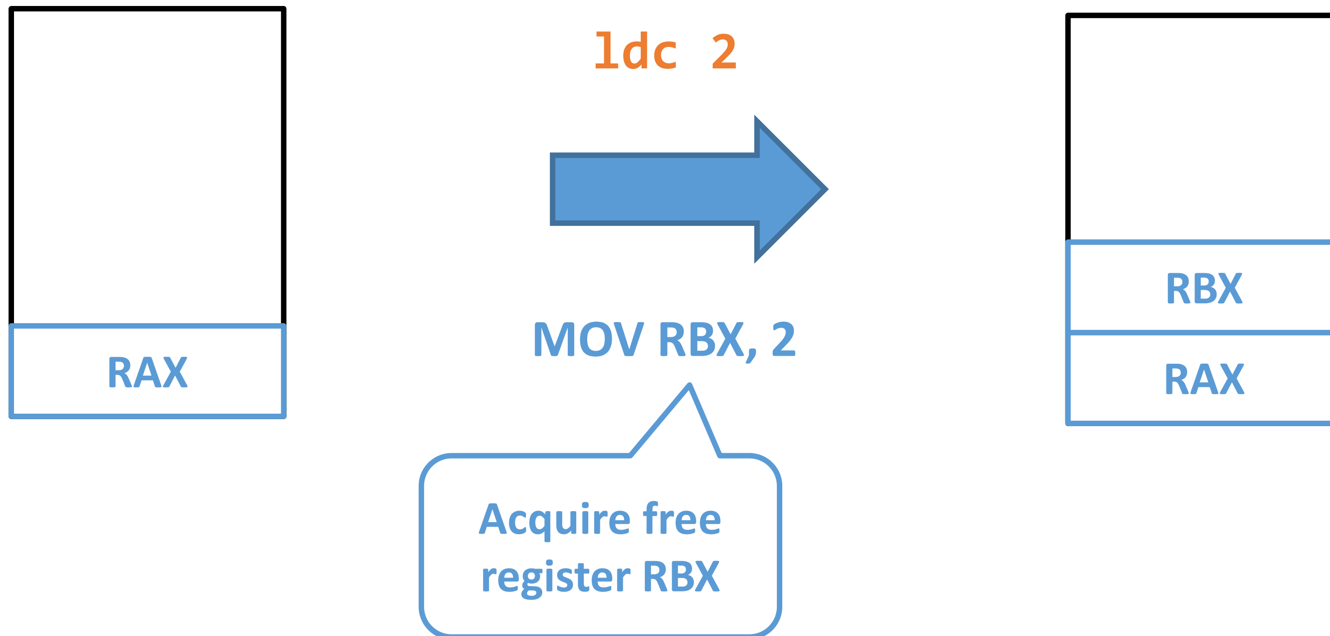
Translation



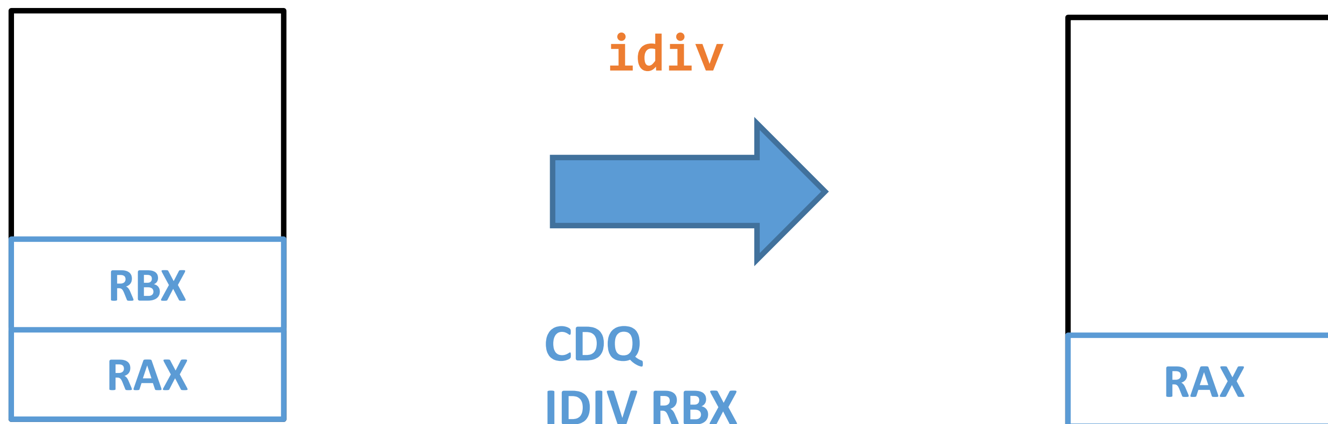
Translation



Translation



Translation



Pop RBX = divisor
Pop RAX = dividend
Push RAX = result

What to do if the dividend is not in RAX?

Register Clobbering

- Backup in temporary register at instructions with fix operands, e.g. IDIV

```
MOV <temp_reg>, RAX
MOV RAX, <reg0>            Backup RAX
CDQ
IDIV <reg1>
MOV <reg0>, RAX            Restore RAX
MOV RAX, <temp_reg>
```

Temporary Backup & Restore

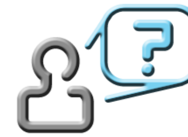
idiv

RCX
RBX
RAX

RBX
RAX

Free temporary register R8

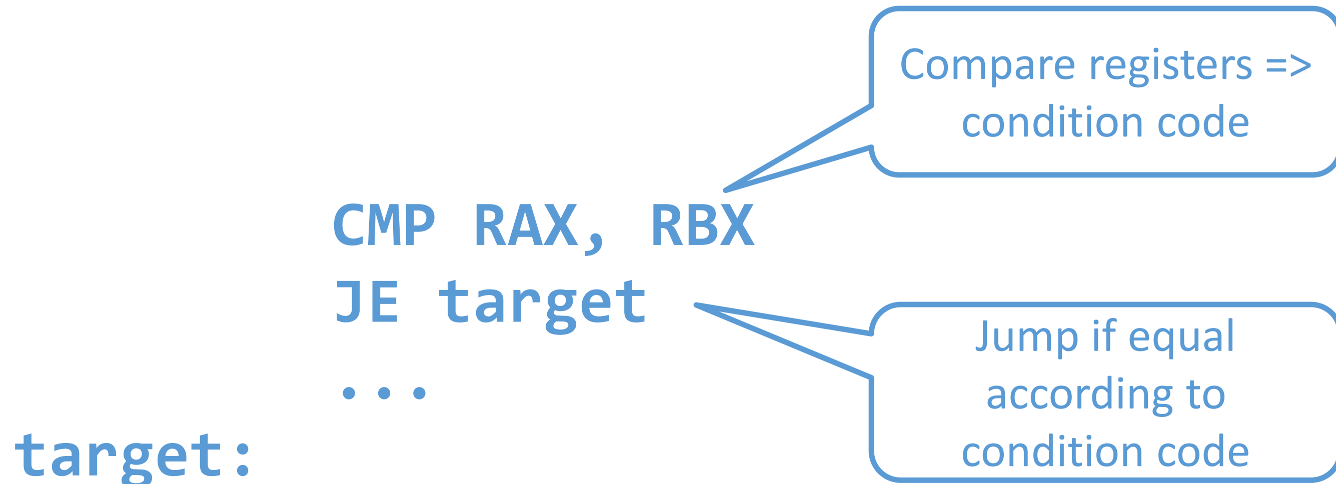
```
MOV R8, RAX
MOV RAX, RBX
CDQ
IDIV RCX
MOV RBX, RAX
MOV RAX, R8
```



What else to consider?

Intel Branches

- Conditional branches based on “condition code”
- Condition code from preceding comparison



Branch Instructions

`CMP reg1, reg2`	Compare
`JE label`	Jump if equal
`JNE label`	Jump if not equal
`JG label`	Jump if greater (reg1 > reg2)
`JGE label`	Jump if greater equal (reg1 >= reg2)
`JL label`	Jump if less (reg1 < reg2)
`JLE label`	Jump if less (reg1 <= reg2)
`JMP label`	Unconditional jump

Relative byte addresses to the labels

Branch Code Template

Instruction pair

icmplt
if_true <target>

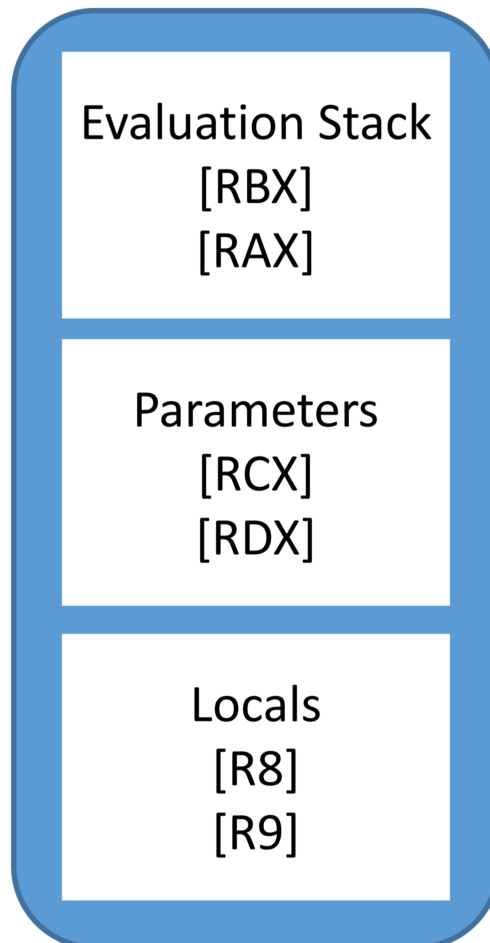
```
// register stack: … -> left -> right
CMP <left>, <right>
JL <offset>
```

Global Register Allocation

- Store variables, especially locals, params in registers
  - Consider register pressure later
- Params are often passed as registers
  - Windows C calling convention:
    RCX, RDX, R8, R9 for first 4 (long) integer parameters
  - Linux and MacOS:
    RDI, RSI, RDX, RCX, R8, R8 for first 6 int parameters
- Idea: Bookkeep current allocation

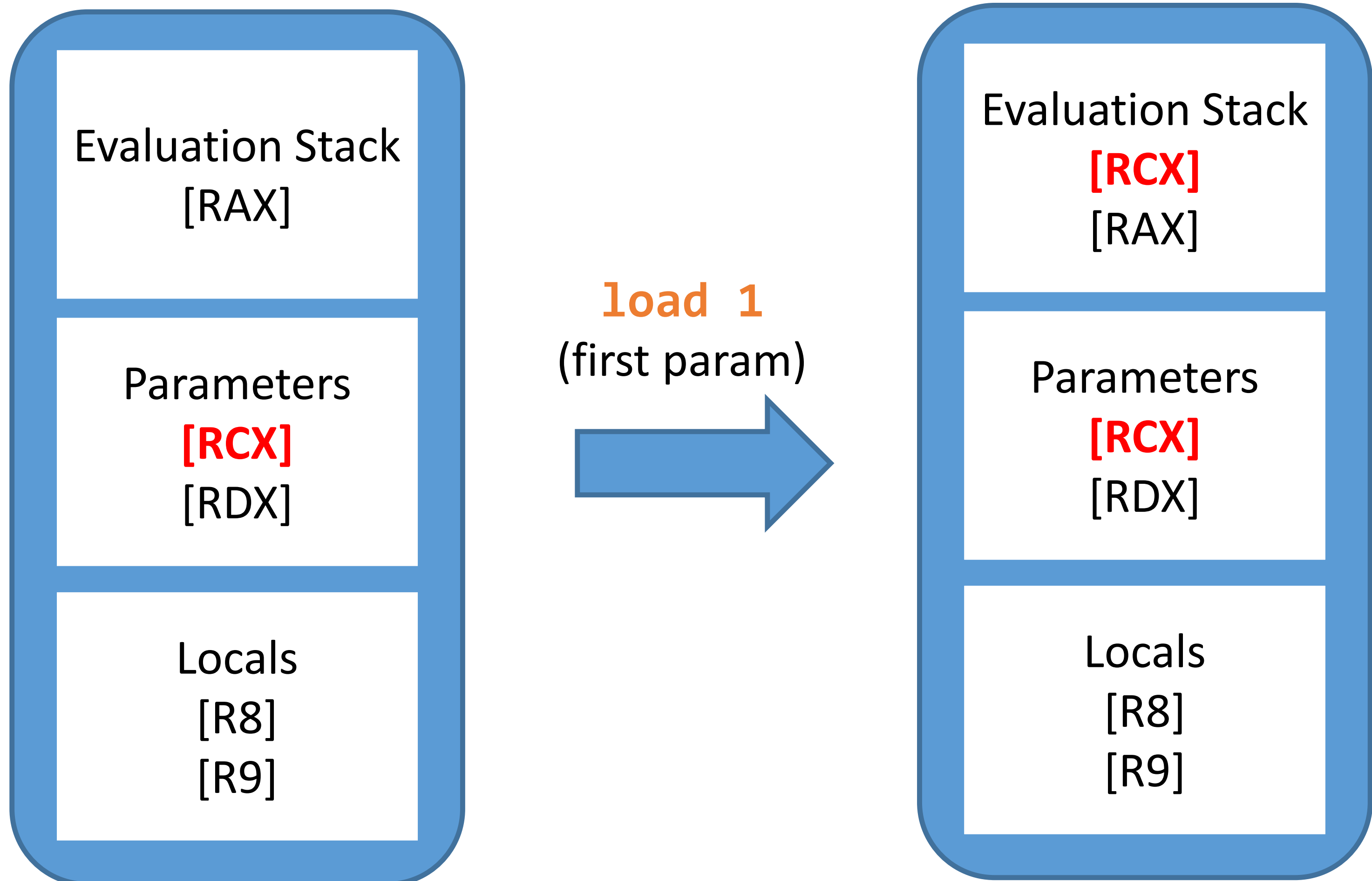
Allocation Record

- Bookkeeping where values are stored at some instruction in the program (in register or memory)



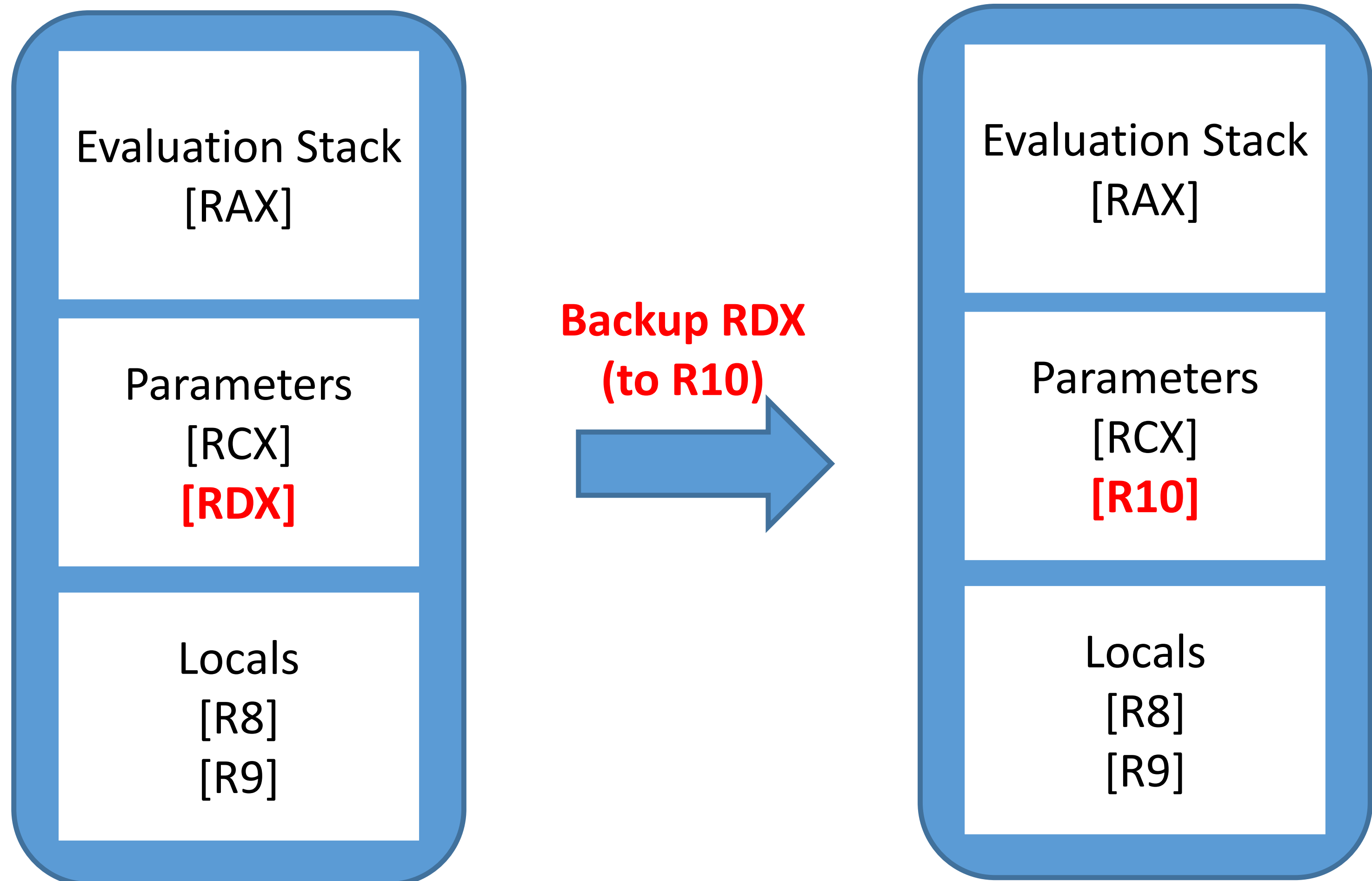
Loading Variables

- Just recycle register in evaluation stack



Backup Values

- Just relocate value, no later restore needed



Register Bookkeeping

Operation	Effect
`acquire()`	Reserve any free register
`release(reg)`	Free register, if not used for locals or parameters
`forceStack(pos, reg)`	Ensure that stack position is in specific register (by relocation if needed)
`reserve(reg)`	Reserve specific register (by relocation if needed)

JIT Compiler Code (ldc)

```
switch (opCode) {
  …
  case LDC:
    var target = acquire();
    var value = (int)instruction.getOperand();
    assembler.MOV_RegImm(target, value);
    push(target);
    break;
  …
}
```

Also handle other constant types (e.g. boolean)

JIT Compiler Code (load)

```
switch (opCode) {
  …
  case LOAD:
    var index = (int)instruction.getOperand();
    /* if parameter index */
    reg = allocation.getParameters().get(index – 1);
    push(reg);
    break;
  …
}
```

Also support local variable reads (index > nofParams)

JIT Compiler Code (store)

```
switch (opCode) {
  …
  case STORE:
    var source = pop();
    var target = allocation.getLocal(index – 1 – nofParams);
    assembler.MOV_RegReg(target, source);
    release(source);
    break;
  …
}
```

Also support parameter writes (index <= nofParams)

JIT Compiler Code (isub)

```
switch (opCode) {
  case ISUB:
    var operand2 = pop();
    var operand1 = pop();
    var result = acquire();
    assembler.MOV_RegReg(result, operand1);
    assembler.SUB_RegReg(result, operand2);
    release(operand1);
    release(operand2);
    push(result);
    break;
  …
}
```

JIT Compiler Code (idiv)

```
switch (opCode) {
  case IDIV:
    reserve(RAX);
    reserve(RDX);
    forceStack(1, RAX);
    var operand2 = pop();
    pop(); // is RAX
    assembler.CDQ();
    assembler.IDIV(operand2);
    push(RAX);
    release(operand2);
    release(RDX);
    break;
  …
}
```

Free RDX:RAX

Dividend must be in RAX

Keep result

Discard RDX

Open Aspects

- Branches (consistent allocations)
- Register pressure
- Native code execution

Next lecture

Review: Learning Goals

- ✓ Understand the purpose and functionality of a Just-In-Time compiler
- ✓ Have an overview of the essential part of the Intel 64 instruction set
- ✓ Know how to implement a first part of the JIT-compiler for your virtual machine

Further Reading

- Dragon book, Register allocation
  – Section 8.8: Global register allocation
  – Section 8.9.2: Template-based code gen (already discussed in week 5)
- Optional, if interested
  – Dragon book, section 8.10: Local register allocation (Ershov numbers)
  – Intel 64 Architecture Specification
    - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html