Recitation 3

Code Generation and x86 Assembly

Explain the previous lectures in plain Java

- Code generation – From IR to CFG
- IR
  - Tree like structure
  - Two major types of nodes: Expression and Statement
  - Nodes can have subnodes recursively (Sub-expressions or sub-statements)
  - Should still correspond to the input file (scope mapped to subnodes)
- CFG
  - Direct cyclic graph of Basic Blocks
  - Each Basic Block is a list of instructions
  - Instructions not necessary identical to assembly instructions
  - Should have no sub-structure
    - Each instruction takes some operands, and operands must be imm, reg, or mem
    - Only branch at the end of a Basic Block

Explain the previous lectures in plain Java

- Use Visitor Pattern (6.005)
- Case 1: Add and Assign
    - Decaf: `y = a + 1;`
- Style 1: Keep track of `returnVar`

```
protected void visit(Add node) {
    Variable left = this.compile(node.left);
    Variable right = this.compile(node.right);

    this.returnVar = new Variable();
    this.currentBasicBlock.add(
        new Instruction(this.returnVar, Op.ADD,
left, right)
    );
}
```

```
protected void visit(Assign node) {
    Variable value = this.compile(node.value);
    Variable var = node.var;

    this.currentBasicBlock.add(
        new Instruction(var, Op.MOV, value)
    );
}
```

Output:

```
TEMP_1 = ADD a, $1
y      = MOV TEMP_1
```

Take care of this in optimization (Copy Propagation)

Explain the previous lectures in plain Java

- Use Visitor Pattern (6.005)
- Case 1: Add and Assign
  - Decaf: `y = a + 1;`
- Style 2: Keep track of `returnVar`, `assignTarget`

```
protected void visit(Add node) {
    if (this.assignTarget != null) {
        this.returnVar = this.assignTarget;
        this.assignTarget = null;
    } else {
        this.returnVar = new Variable();
    }

    Variable left = this.compile(node.left);
    Variable right = this.compile(node.right);
    this.currentBasicBlock.add(
        new Instruction(this.returnVar, Op.ADD,
left, right)
    );
}
```

```
protected void visit(Assign node) {
    this.assignTarget = node.var;
    this.compile(node.value);
}
```

Output:

```
y       = ADD a, $1
```

Explain the previous lectures in plain Java

- Use Visitor Pattern (6.005)
- Case 2: If Statement
  - Decaf: `if (a || b) {t} else {f};`
- Keep track of `returnVar`, `trueTarget`, `falseTarget`

```
protected void visit(Or node) {
    if (this.trueTarget != null) {
        // This bool expr is being evaluated
        BasicBlock right = new BasicBlock();
        BasicBlock currentTrue = this.trueTarget;
        BasicBlock currentFalse = this.falseTarget;
        this.falseTarget = right;
        this.compile(node.left);

        this.currentBasicBlock = right;
        this.trueTarget = currentTrue;
        this.falseTarget = currentFalse;
        this.compile(node.right);
    } else {
        // This bool expr is being assigned
        ...
    }
}
```

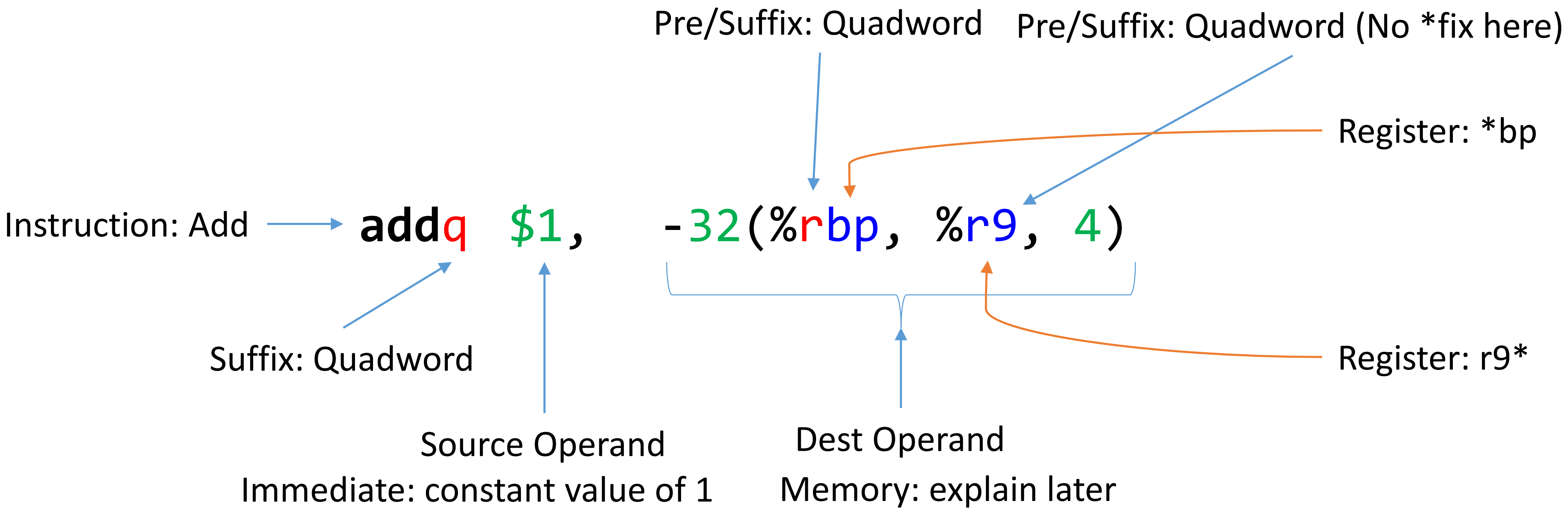
```
protected void visit(If node) {
    BasicBlock t = new BasicBlock();
    BasicBlock f = new BasicBlock();
    BasicBlock exit = new BasicBlock();

    this.trueTarget = t;
    this.falseTarget = f;
    this.compile(node.cond);
    this.trueTarget = null;
    this.falseTarget = null;

    this.currentBasicBlock = t;
    this.visit(node.trueStatement);
    this.currentBasicBlock.add(
        new Instruction(null, Op.JMP, exit)
    );
    ...
}
```

Time to Assemble!

- Basics
- Instructions that you are going to use
- Instructions that you should never use
- Calling convention
- Gotchas and tricks

Parts of an Instruction

- We are using gcc syntax (also known as AT&T syntax) in 6.035
- If you are reading the Intel manual (!), please reverse the order of the operands.



Size Prefixes/Suffixes

For Instruction

Name	Size (bits)	Suffix
Byte	8	b
Word	16	w
Doubleword	32	l
Quadword	64	q

Most instructions (!) can take a suffix, even some make no sense and have no effect, for example `retq`. In this case you don't need one.

For Operand

8-bit GP	16-bit GP	32-bit GP	64-bit GP
al	ax	eax	rax
cl	cx	ecx	rcx
dl	dx	edx	rdx
bl	bx	ebx	rbx
spl	sp	esp	rsp
bpl	bp	ebp	rbp
sil	si	esi	rsi
dil	di	edi	rdi
r8l	r8w	r8d	r8
r9l	r9w	r9d	r9
r10l	r10w	r10d	r10
r11l	r11w	r11d	r11
r12l	r12w	r12d	r12
r13l	r13w	r13d	r13
r14l	r14w	r14d	r14
r15l	r15w	r15d	r15

Same row share the same register space
For example, modifying %al changes the lower 8 bit value of %rax

Operand types

- Immediate = Constant

  $1234

  - Sign extended
  - Size types: imm8, imm16, imm32, and imm64 (!)
  - Type is automatically derived based on instruction suffix*
  - However only **MOV** instruction can take imm64
  - All other instructions can only take imm32

- Register

  %rax

  - Size type is self-indicated
  - Make sure the size doesn't contradict with the size of the instruction

- Memory
  - Absolute 0xdeadbeef
    - Never use
  - rip relative a(%rip)
    - Can be short-handed as a
    - For global objects
  - Indirect
    - All components

```
        1234(%rax, %rdi, 4)
displacement(base, index, multiplier)
= *(base + index * multiplier + displacement)
```

    - Multiplier is one of {1, 2, 4, 8} (default = 1)
    - Displacement is imm32 (default = 0)
    - Handy for array access
    - Some components can be missing

```
1234(%rax, %rdi, 4) = *(%rax)
1234(%rax, %rdi, 4) = *(%rax + $1234)
1234(%rax, %rdi, 4) = *(%rax + %rdi * 1)
1234(%rax, %rdi, 4) = *(%rax + %rdi * 4)
1234(%rax, %rdi, 4) = *($1234 + %rdi * 4)
```

Valid combination of operands

For most two-operand instructions

Src \ Dest	Imm32	Reg	Mem
Imm32	✗	✓	✓
Reg	✗	✓	✓
Mem	✗	✓	✗

Load/Store ??

- There is no instruction for load or store
- Use memory operand for that!

Status Flags

- Every arithmetic instruction sets flags (in %rflags)
- Every conditional jump instruction reads (1 or more) flag(s) and jump if those flags are set to 1
- Jump instructions that you will use: je, jne, jg, jge, jl, jle, and they correspond to the 6 comparison operations.
  - However, they only (really) correspond if the last arithmetic instruction you performed is sub or cmp
  - For example

```
if ( x < y ) {
  true_stmt …
} else {
  false_stmt …
}
```

```
        cmpq    y,      x       # x - y
        jge     .L2
                true_stmt …
        jmp     .L3
.L2
                false_stmt …
        jmp     .L3
.L3
```

- See https://pdos.csail.mit.edu/6.828/2016/readings/i386/appc.htm

Declaring functions, strings, and global vars

Functions

No more type information in assembly

Decaf: `void main() {...}`

```
        .text
        .global main
        .type   main, @function
main:
        ...
```

Declaring functions, strings, and global vars

Strings

Decaf: printf("hello, world“);

```
        .section        .rodata
str_0:
        .string "hello, world"
```

Using a string

```
# inside a function

        ...

        movq    $0,     %rax # explain later
        movq    $str_0, %edi
        call    printf
```

Declaring functions, strings, and global vars

Globals

Decaf: `int a[10];`

```
        .globl  a
        .bss
        .align  32 # optional
        .type   a, @object
        .size   a, 80 # size = 8*10
a:
        .zero   80
```

Using a global

Decaf: `a[i] = 2;`

Decaf: `printf("hello, world")`

```
# inside a function

        ...
        # assume the value of i is in %rax
        movq    $2,     a(, %rax, 8)
```

Time to Assemble!

- Basics
- Instructions that you are going to use
- Instructions that you should never use
- Calling convention
- Gotchas and tricks

Instructions that you are going to use (probably)

- Arithmetics
  - Regular: add, sub, imul, idiv, neg, cmp, test
  - Probably (due to optimizations): xor, sal, sar, inc, dec, lea
- Move
  - Regular: mov, movsx, push, pop, cqo
  - Probably (different IR design): set**
  - Probably (optimization): cmov**
- Control flow
  - Regular: call, jmp, j**, ret
- Misc
  - Regular: nop

Time to Assemble!

- Basics
- Instructions that you are going to use
- Instructions that you should never use
- Calling convention
- Gotchas and tricks

Instructions that you should NOT use

- Unless you know what you are doing
- Arithmetic
  - Any ASCII/BCD related arith op
  - Any Floating Point op
  - Unsigned mul and div (Not to be confused with imul and idiv)
    - We use signed integer in decaf, and never use unsigned integer
  - and, or, not – these are bitwise operations, not boolean opeartions (see slide 5)
- Move
  - xchg – slow, implicit lock if one operand is mem
  - Flag manipulation op – you should treat %rflags as a black box.
- Procedural Call
  - enter, leave – too slow, just don't use. Recall 6.004 How to adjust stack
- Control flow
  - Any complex op, like repz prefix
  - bound – wrong way to do bounds checking
  - syscall, int – wrong way to call external functions, libc functions are enough

Time to Assemble!

- Basics
- Instructions that you are going to use
- Instructions that you should never use
- Calling convention
- Gotchas and tricks

Calling convention (The “C” Convention)

- Arguments order - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Rest (if any) - push to stack from **RIGHT** to **LEFT**
  - 6.035 Decaf Spec requires arguments themselves should be **evaluated** from **left** to **right**
- Return - %rax
- Preserved across function calls: %rbx, %rsp, %rbp, %r12-%r15
- For integral-typed values only, including pointers (and don’t worry about FP values)

Calling convention (The “C” Convention)

- Gotcha 1 – variadic function
  - Any external C function with the name like this: *printf*, and *scanf*, and does not start with ‘v’, is variadic
  - For variadic function call, %rax indicates how many FP arguments
  - Therefore you need to set %rax to 0 before calling functions like these
  - May causes SEGMENTATION FAULT if you don’t do so
- There is no other variadic function in libc, besides those mentioned above
- There is no variadic function in Decaf

Calling convention (The “C” Convention)

- Gotcha 2 – 16 byte alignment
  - Right before a function call (`call` instruction) , the value of %rsp must be a multiple of 16
  - Right after the call instruction %rip is pushed to stack (%rsp -= 8)
    - Formula for stack reserve
      [Actual adjust to rsp] = 8 + ceil(([stack space you need for local var] – 8) / 16) * 16
  - Stack is aligned before main by the OS, so you need to preserve this invariant
  - Only matters if involving some libc functions
    - Mostly those invoking a syscall, if you know which ones will
  - May causes SEGMENTATION FAULT if you don’t do so
  - Doesn’t matter if there is no libc call in the call graph.

Calling convention (The “C” Convention)

- 128 Byte scratch zone below stack
  - Enforced by OS
  - Don’t need to adjust %rsp in a leaf function
  - Useful if you have a register spill

Time to Assemble!

- Basics
- Instructions that you are going to use
- Instructions that you should never use
- Calling convention
- Gotchas and tricks

Add

- add instruction, of course
- There is another instruction to do addition – Load Effective Address
- lea mem, reg – calculate the address value of mem, and store to reg
    - `lea (reg1, reg2, mult=1), reg3` → `reg3 = reg1 + reg2 * mult`
    - `lea imm(reg1), reg3` → `reg3 = imm + reg1`
    - `lea imm(reg1, reg2, mult=1), reg3` → `reg3 = imm + reg1 + reg2 * mult`
    - `lea imm(, reg2, mult=1), reg3` → `reg3 = imm + reg2 * mult`
    - mult is {1, 2, 4, 8}, default is 1
- Useful when you don’t want to modify the value of the augend
- Mostly used for optimizations

Subtract

`sub a, b`

b = b - a

`cmp a, b`

b - a (only set flags, discards result)

Note that this is reversed in Intel handbook, because different syntax

To identify which syntax a manual is using, check if it ever makes imm as the second operand, in this case it is Intel syntax. The opposite case means it is GCC syntax.

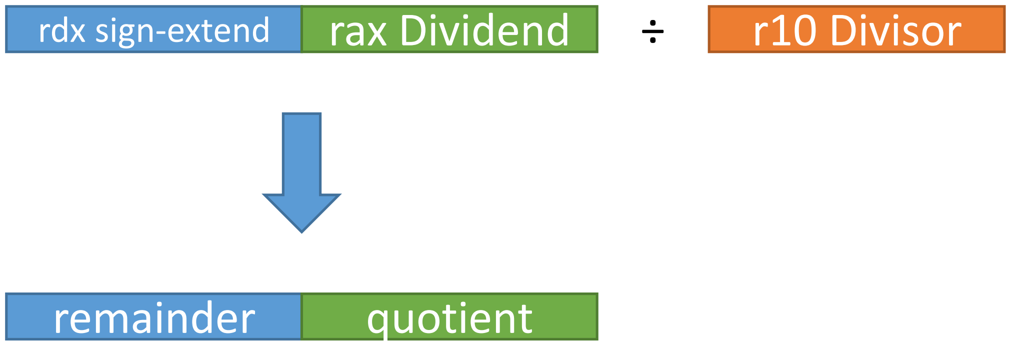
Multiply

- Do NOT use mul instruction
- Use imul instead
- Destination operand can only be reg
- Two formats
  - `imul reg/mem, reg` (without imm)
    - `imul a, b` → `b = a * b`
  - `imul imm, reg/mem, reg` (with imm)
    - `imul imm, a, b` → `b = a * imm`

Divide

- Both divide and mod use the same instruction – `idiv`
  - Do NOT use div
- Implicitly reads and writes %rax and %rdx
- %rdx:%rax (128 bit) is dividend, takes reg or mem as divisor (no imm)
  - Cannot reuse %rax or %rdx as divisor, otherwise you may get floating point error
- %rax is quotient, %rdx is remainder
- Use cqo instruction after you move the 64-bit dividend to %rax, before idiv instruction
  - Since you only have 64-bit int, cqo instruction performs sign extension to %rax, and puts the high 64 bits to %rdx
  - Don't just set %rdx to 0, you get wrong results for negative numbers
- Very slow instruction (30-50 cycles typical)

```
movq   Dividend,   %rax
movq   Divisor,    %r10
cqo
idivq %r10
```



Note: Do not check for division by 0 in Decaf

How many branches does it take to check the range?

How many branches does it take to check the range?

- Answer: 1
- Unsigned int hack

```
int a[10];
a[i] = 3;
```

Assume i is in %rax

```
cmpq $10, %rax
jae  .handle_error
movq $3, ...
```

How to terminate the program immediately

```
movq [whatever exit status], %rdi
call exit
```

Gotcha: Boolean arguments

- A bool value only occupy partial of a register
- High bits can be arbitrary junk values
- May led to wrong result if you are calling libc functions, since booleans are treated as integers in C
- Solution – use `movzx`
  - Zero extends bool to int (fill high bits with 0)

Lower from 3-operand IR to 2-operand asm

- 3-operand IR: c = op a b
- Unoptimized code generation
  - `mov a, %r10`
  - `mov b, %r11`
  - `op %r10, %r11`
  - `mov %r11, c`
- Always work, regardless a, b, and c are reg/mem/imm (c cannot be imm)

Lower from 3-operand IR to 2-operand asm

- 3-operand IR: c = op a b
- Optimized codegen? (you have to do it anyways for Project 5)
- You need to consider the following: For every case in the product space of all these possible combinations (Most cases can be combined)
  - a in {reg, mem, imm32, imm64} ?
  - b in {reg, mem, imm32, imm64} ?
  - c in {reg, mem} ?
  - a == c? b == c? a == b?
  - Is op communicative?