CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework:
Basic Blocks

More Assembly:
Zero Extend / Sign Extend

Register Aliases

Setting the 32-bit view *clears* the top 32 bits. Setting the 16- or 8-bit views does *not*.

There is no equivalent to AH, BH, CH, and DH for the other registers.

Setting the low 32 bits is *zero extended* into the top 32 bits.

64-bit Register	Lowest 32 Bits	Lowest 16 Bits	Lowest 8 Bits
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Zero-Extend Move

So `mov rax, bl` does not zero out the top bits...

Zero-Extend Move

So `mov rax, bl` does not zero out the top bits...

But `movzx rax, bl` *does*. This is a zero-extended move. The destination is always a register. The source can be a register or memory.

- moving a byte (register or memory) to a word, double word, or quadword register
  `mov eax, r11b`
- moving a word (register or memory) to a double word or quadword register
  `mov rax, WORD [rbp+0x20]`

Very, very common: 5,651 in the `python3.7` executable alone.

Zero-Extend Move

But what if the value you are moving is a *signed value*?

Sign-Extend Move

But what if the value you are moving is a *signed value*?

Use the sign-extended move: `movsx rax, bl`. This is a sign-extended move. The destination is always a register. The source can be a register or memory. The result it the sign-extended value, so moving a byte value of -17 (`0xef`) sign-extended into a double word register is still -17 (`0xffffffef`).

Very, very common: 2,392 in the `python3.7` executable alone.

Unlike `movzx`, there is a version to move a double word to a quadword with sign extension (this is different from the automatic zero extension): `movsxd`

Slicing

Slicing

Program slicing is a method of *simplifying* a program to make analysis simpler by focusing on a particular aspect of *program semantics* called the **slicing criterion**.

Given a location *l* and a variable *v*, a **slice** is constructed with respect to (*l*,*v*) by deleting all statements irrelevant to the value of *v* at *l*.

See:

Harman, M., and Hierons, R., "An Overview of Program Slicing," *Software Focus*, vol. 2, no. 3 (2001): 85–92. https://doi.org/10.1002/swf.41.

Forward and Backward Slicing

Backward Slicing

A **backward slice** consists of statements that have an *effect on* the slicing criterion.

Forward Slicing

A **forward slice** consists of statements that are *affected by* the slicing criterion.

Forward and Backward Slicing

Backward Slicing

A **backward slice** consists of statements that have an *effect on* the slicing criterion.

(We only consider backward slicing here.)

Forward Slicing

A **forward slice** consists of statements that are *affected by* the slicing criterion.

Backward (Static) Slicing

```
x = 1;
y = 2;
z = y - 2;
r = x;
z = x + y;          // Slice point
```

Construct the slice for variable z at the end of the program.

Backward (Static) Slicing

```
x = 1;
y = 2;
z = y - 2;
r = x;
z = x + y;          // z depends on x
and y
```

Construct the slice for variable z at the end of the program.

Backward (Static) Slicing

```
x = 1;
y = 2;
z = y - 2;
~~r = x;~~                    // does not modify
x,y
z = x + y;
```

Construct the slice for variable z at the end of the program.

Backward (Static) Slicing

x = 1;
y = 2;
~~z = y - 2;~~ // does not modify x,y
~~r = x;~~
z = x + y;

Construct the slice for variable z at the end of the program.

Backward (Static) Slicing

```
x = 1;
y = 2;                    // modifies y
~~z = y - 2;~~
~~r = x;~~
z = x + y;
```

Construct the slice for variable z at the end of the program.

Backward (Static) Slicing

```
x = 1;              // modifies x
y = 2;
~~z = y - 2;~~
~~r = x;~~
z = x + y;
```

Construct the slice for variable z at the end of the program.

Backward (Static) Slicing

```
x = 1;
y = 2;
z = x + y;
```

Construct the slice for variable z at the end of the program.

Example

```c
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int s = 0;      // Sum
    int p = 0;      // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p);
    return 0;
}
```

Print the *sum* and *product* of the sequence of integers from 1 up to *n*.

Example

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int s = 0;      // Sum
    int p = 0;      // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p);
    return 0;
}
```

Print the *sum* and *product* of the sequence of integers from 1 up to *n*.

```
$ sumprod
5
15, 0
```

Something is wrong with the product.

Example: Static Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int s = 0;      // Sum
    int p = 0;      // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p); // Want p here.
    return 0;
}
```

Print the *sum* and *product* of the sequence of integers from 1 up to *n*.

```
$ sumprod
5
15, 0
```

Something is wrong with the product.

Example: Static Slicing

```c
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int s = 0;          // Sum
    int p = 0;          // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p); // Want p here.
    return 0;
}
```

Print the *sum* and *product* of the sequence of integers from 1 up to *n*.

```
$ sumprod
5
15, 0
```

Something is wrong with the product.

Example: Static Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int p = 0;      // Product
    while (n > 0) {
        p *= n;
        --n;
    }
}
```

Print the *sum* and *product* of the sequence of integers from 1 up to *n*.

```
$ sumprod
5
15, 0
```

Something is wrong with the product: `p` should be initialized to one, not zero.

The slice is *most of the program*. Most well-written programs are *cohesive*, so you get large(ish) slices.

Dynamic Slicing

```c
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int s = 0;      // Sum
    int p = 0;      // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p);
    return 0;
}
```

We can add information about runtime values.
This is called **dynamic** slicing.

Example: Dynamic Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int s = 0;      // Sum
    int p = 0;      // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p);
    return 0;
}
```

We can add information about runtime values. This is called **dynamic** slicing.

For example, let's assume n is zero. The product should be one.

Example: Dynamic Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int s = 0;        // Sum
    int p = 0;        // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p);
    return 0;
}
```

We can add information about runtime values. This is called **dynamic** slicing.

For example, let's assume n is zero. The product should be one.

Example: Dynamic Slicing

```
#include <stdio.h>

int main( void ) {
    int p = 0;      // Product
}
```

We can add information about runtime values. This is called **dynamic** slicing.

For example, let's assume n is zero. The product should be one.

The slice is much smaller… and the problem is obvious.

Constraint Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    if (n <= 0) {
        return 1;
    }
    int s = 0;      // Sum
    int p = 1;      // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p);
    return 0;
}
```

Other kinds of slicing are possible.

An interesting kind is **constraint slicing**. Here we give a constraint on the values expected at runtime, and then slice.

This can be hard (quantification) or relatively easy (Pressburger arithmetic).

Example: Constraint Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    if (n <= 0) {
        return 1;
    }
    int s = 0;      // Sum
    int p = 1;      // Product
    while (n > 0) {
        s += n;
        p *= n;
        --n;
    }
    printf("%d, %d\n", s, p);
    return 0;
}
```

Other kinds of slicing are possible.

An interesting kind is **constraint slicing**. Here we give a constraint on the values expected at runtime, and then slice.

This can be hard (quantification) or relatively easy (Pressburger arithmetic).

Assume `n`>1. Still considering `p`.

Example: Constraint Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
            n>1
    scanf( "%d", &n );                       n>1
    if (n <= 0) {
            n>1
        return 1;
            n>1 && n<=0
    }
    int s = 0;      // Sum                   n>1
    int p = 1;      // Product               n>1
    while (n > 0) {

        s += n;
            n>1 && n>0
        p *= n;
        --n;
            n+1>1 && n+1>0
```

Other kinds of slicing are possible.

An interesting kind is **constraint slicing**. Here we give a constraint on the values expected at runtime, and then slice.

This can be hard (quantification) or relatively easy (Pressburger arithmetic).

Assume n>1. Still considering p.

Example: Constraint Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
             n>1
    scanf( "%d", &n );                        n>1
    if (n <= 0) {
             n>1
        return 1;
             n>1 && n<=0
    }
    int s = 0;      // Sum                    n>1
    int p = 1;      // Product                n>1
    while (n > 0) {

        s += n;
             n>1 && n>0
        p *= n;
        --n;
             n+1>1 && n+1>0
```

Other kinds of slicing are possible.

An interesting kind is **constraint slicing**. Here we give a constraint on the values expected at runtime, and then slice.

This can be hard (quantification) or relatively easy (Pressburger arithmetic).

Assume n>1. Still considering p.

Example: Constraint Slicing

```
#include <stdio.h>

int main( void ) {
    int n;
    scanf( "%d", &n );
    int p = 1;      // Product
    do {
        p *= n;
        --n;
    } while (n > 0); // Loop unrolled once
}
```

Other kinds of slicing are possible.

An interesting kind is **constraint slicing**. Here we give a constraint on the values expected at runtime, and then slice.

This can be hard (quantification) or relatively easy (Pressburger arithmetic).

Assume n>1. Still considering p.

Can be used to optimize programs based on input.

Slicing Assembly

Slicing Assembly

What do we slice?

- **Assembly**
  Starts and ends with assembly. Can be tricky!
- **Semantics**
  Might not end with assembly, or might have to invent new assembly.

```
inc rax
lea rcx, [rax*8]
push rcx
push rax
mov rdi, 21
call _optc
pop rcx
pop rax
; want to know rax here
```

Slicing Assembly

What do we slice?

- **Assembly**
  Capstone can tell us which registers are *read* and which are *written*.

```
inc rax
lea rcx, [rax*8]
push rcx
push rax
mov rdi, 21
call _optc
pop rcx
pop rax
; want to know rax here
```

Example: Slicing Assembly

Instruction	Read	Written
`inc rax`	`rax`	`rax`
`lea rcx, [rax*8]`	`rax`	`rcx`
`push rcx`	`rsp, rcx`	
`push rax`	`rsp, rax`	
`mov rdi, 21`		
`call _optc`	`rax, rdi`	
`pop rcx`	`rsp, M`	`rsp, rcx`
`pop rax`	`rsp, M`	`rsp, rax`

If something we need is written (lvalue) then we need the things that are read (rvalue). Note this is approximate without more semantic information!

Example: Slicing Assembly

Instruction	Read	Written	Depends (for rax)
inc rax	rax	rax	
lea rcx, [rax*8]	rax	rcx	
push rcx	rsp, rcx	rsp, M	
push rax	rsp, rax	rsp, M	
mov rdi, 21		rdi	
call _optc	rax, rdi	rax, rcx	
pop rcx	rsp, M	rsp, rcx	
pop rax	rsp, M	rsp, rax	rsp, M
			rax

Example: Slicing Assembly

Instruction	Read	Written	Depends (for `rax`)
`inc rax`	`rax`	`rax`	
`lea rcx, [rax*8]`	`rax`	`rcx`	
`push rcx`	`rsp, rcx`	`rsp, M`	
`push rax`	`rsp, rax`	`rsp, M`	
`mov rdi, 21`		`rdi`	
`call _optc`	`rax, rdi`	`rax, rcx`	
`pop rcx`	`rsp, M`	`rsp, rcx`	`rsp, M`
`pop rax`	`rsp, M`	`rsp, rax`	`rsp, M`
			`rax`

Example: Slicing Assembly

Instruction	Read	Written	Depends (for `rax`)
`inc rax`	`rax`	`rax`	
`lea rcx, [rax*8]`	`rax`	`rcx`	
`push rcx`	`rsp, rcx`	`rsp, M`	
`push rax`	`rsp, rax`	`rsp, M`	`rsp, rax`
`mov rdi, 21`		`rdi`	`rsp, M`
`call _optc`	`rax, rdi`	`rax, rcx`	`rsp, M`
`pop rcx`	`rsp, M`	`rsp, rcx`	`rsp, M`
`pop rax`	`rsp, M`	`rsp, rax`	`rsp, M`
			`rax`

Example: Slicing Assembly

Instruction	Read	Written	Depends (for rax)
inc rax	rax	rax	
lea rcx, [rax*8]	rax	rcx	
push rcx	rsp, rcx	rsp, M	rsp, rcx, rax
push rax	rsp, rax	rsp, M	rsp, rax
mov rdi, 21		rdi	rsp, M
call _optc	rax, rdi	rax, rcx	rsp, M
pop rcx	rsp, M	rsp, rcx	rsp, M
pop rax	rsp, M	rsp, rax	rsp, M
			rax

Example: Slicing Assembly

Instruction	Read	Written	Depends (for `rax`)
`inc rax`	`rax`	`rax`	
`lea rcx, [rax*8]`	`rax`	`rcx`	`rsp, rax`
`push rcx`	`rsp, rcx`	`rsp, M`	`rsp, rcx, rax`
`push rax`	`rsp, rax`	`rsp, M`	`rsp, rax`
`mov rdi, 21`		`rdi`	`rsp, M`
`call _optc`	`rax, rdi`	`rax, rcx`	`rsp, M`
`pop rcx`	`rsp, M`	`rsp, rcx`	`rsp, M`
`pop rax`	`rsp, M`	`rsp, rax`	`rsp, M`
			`rax`

Example: Slicing Assembly

Instruction	Read	Written	Depends (for rax)
inc rax	rax	rax	rsp, rax
lea rcx, [rax*8]	rax	rcx	rsp, rax
push rcx	rsp, rcx	rsp, M	rsp, rcx, rax
push rax	rsp, rax	rsp, M	rsp, rax
mov rdi, 21		rdi	rsp, M
call _optc	rax, rdi	rax, rcx	rsp, M
pop rcx	rsp, M	rsp, rcx	rsp, M
pop rax	rsp, M	rsp, rax	rsp, M
			rax

Example: Slicing Assembly

Instruction	Read	Written	Depends (for `rax`)
`inc rax`	`rax`	`rax`	`rsp, rax`
`lea rcx, [rax*8]`	`rax`	`rcx`	`rsp, rax`
`push rcx`	`rsp, rcx`	`rsp, M`	`rsp, rcx, rax`
`push rax`	`rsp, rax`		
`mov rdi, 21`			
`call _optc`	`rax, rdi`		
`pop rcx`	`rsp, M`	`rsp, rcx`	`rsp, M`
`pop rax`	`rsp, M`	`rsp, rax`	`rsp, M`
			`rax`

Now we look for disjoint sets (if nothing we need is written by an instruction… we don't need the instruction)

Example: Slicing Assembly

Instruction	Read	Written	Depends (for `rax`)	
`inc rax`	`rax`	`rax`	`rsp, rax`	
`lea rcx, [rax*8]`	`rax`	`rcx`	`rsp, rax`	
`push rcx`	`rsp, rcx`	`rsp, M`	`rsp, rcx, rax`	
`push rax`	`rsp, rax`	`rsp, M`	`rsp, rax`	
`mov rdi, 21`		`rdi`	`rsp, M`	
`call _optc`	`rax, rdi`	`rax, rcx`	`rsp, M`	
`pop rcx`	`rsp, M`	`rsp, rcx`	`rsp, M`	
`pop rax`	`rsp, M`	`rsp, rax`	`rsp, M`	needed
			`rax`	

Example: Slicing Assembly

Instruction	Read	Written	Depends (for `rax`)	
`inc rax`	`rax`	`rax`	`rsp, rax`	
`lea rcx, [rax*8]`	`rax`	`rcx`	`rsp, rax`	
`push rcx`	`rsp, rcx`	`rsp, M`	`rsp, rcx, rax`	
`push rax`	`rsp, rax`	`rsp, M`	`rsp, rax`	
`mov rdi, 21`		`rdi`	`rsp, M`	
`call _optc`	`rax, rdi`	`rax, rcx`	`rsp, M`	
`pop rcx`	`rsp, M`	`rsp, rcx`	`rsp, M`	needed
`pop rax`	`rsp, M`	`rsp, rax`	`rsp, M`	
			`rax`	

Example: Slicing Assembly

Instruction	Read	Written	Depends (for rax)	
inc rax	rax	rax	rsp, rax	
lea rcx, [rax*8]	rax	rcx	rsp, rax	
push rcx	rsp, rcx	rsp, M	rsp, rcx, rax	
push rax	rsp, rax	rsp, M	rsp, rax	
mov rdi, 21		rdi	rsp, M	
call _optc	rax, rdi	rax, rcx	rsp, M	not needed
pop rcx	rsp, M	rsp, rcx	rsp, M	
pop rax	rsp, M	rsp, rax	rsp, M	
			rax	

Example: Slicing Assembly

Instruction	Read	Written	Depends (for rax)
inc rax	rax	rax	rsp, rax
lea rcx, [rax*8]	rax	rcx	rsp, rax
push rcx	rsp, rcx	rsp, M	rsp, rcx, rax
push rax	rsp, rax	rsp, M	rsp, rax
mov rdi, 21		rdi	rsp, M
call _optc	rax, rdi	rax, rcx	rsp, M
pop rcx	rsp, M	rsp, rcx	rsp, M
pop rax	rsp, M	rsp, rax	rsp, M
			rax

Example: Slicing Assembly

Instruction	Read	Written	Depends (for rax)
inc rax	rax	rax	rsp, rax
lea rcx, [rax*8]	rax	rcx	rsp, rax
push rcx	rsp, rcx	rsp, M	rsp, rcx, rax
push rax	rsp, rax		
mov rdi, 21			
call _optc	rax, rdi		
pop rcx	rsp, M	rsp, rcx	rsp, M
pop rax	rsp, M	rsp, rax	rsp, M
			rax

At this point you should be able to see what is happening: RCX and RAX are swapped on the stack.

Slicing Assembly

What do we slice?

- **Semantics**
  Might not end with assembly, or might have to invent new assembly.

```
inc rax
lea rcx, [rax*8]
push rcx
push rax
mov rdi, 21
call _optc
pop rcx
pop rax
; want to know rax here
```

Assembly Semantics

x86-64 Semantics

It would be *great* if there were a complete and usable semantics for the x86-64 instruction set!

x86-64 Semantics

It would be *great* if there were a complete and usable semantics for the x86-64 instruction set!

1. Complete in what sense?
2. Usable for what / to whom?

x86-64 Semantics

It would be *great* if there were a complete and usable semantics for the x86-64 instruction set!

1. Complete in what sense? ← What can we do with the semantics.
2. Usable for what / to whom? ← What representation do we need.

x86-64 Semantics

It would be *great* if there were a complete and usable semantics for the x86-64 instruction set!

1. Complete in what sense? ← What can we do with the semantics.
2. Usable for what / to whom? ← What representation we need.

There are multiple semantic dimensions to a program.

- Do we care about *parallelism*? Then we need to know when values are stable, how pipelines are used, etc.
- Do we care about *timing*? Then we need cycle-accurate information about the command, which can vary depending on pipelining, etc.

x86-64 Semantics

It would be *great* if there were a complete and usable semantics for the x86-64 instruction set!

1. Complete in what sense? ← What can we do with the semantics.
2. Usable for what / to whom? ← What representation we need.

There are multiple semantic representations we can use.

- Focus on analysis? Functional semantics are composable.
- Focus on emulation / simulation? Operational semantics are executable.
- Sometimes we can have a bit of both: Represent semantics in a form like Lisp or ML.

Representations

Many forms are possible...

- Conditional concurrent assignments
- Conditional sequential assignments
- Lisp
- k-Expressions
- C source code

```
adc eax, 17 :=
[
  true ->
  [   eax = eax + 17 + bit(intel_CF)
    : intel_ZF = ((eax + 17 + bit(intel_CF)) == 0)
    : intel_SF = ((signed(eax + 17 + bit(intel_CF)) < 0))
    : intel_PF = is_even_parity_lowbyte((eax + 17 + bit(intel_CF)))
    : intel_CF = carry_flag_adc_32(eax, 17, intel_CF)
    : intel_OF = overflow_flag_adc_32(eax, 17, intel_CF)
    : intel_AF = auxiliary_carry_flag_adc(eax, 17, intel_CF)
  ]
];
```

Used in Hyperion
https://www.affirmlogic.com/

Representations

Many forms are possible…

- Conditional concurrent assignments
- Conditional sequential assignments
- Lisp
- k-Expressions
- C source code

```
fe50: addq $0x1, %rax
{
  v18533 := RAX
  RAX := RAX + 1
  CF := RAX < v18533
  OF := ~high:1[v18533] & (high:1[v18533] ^ high:1[RAX])
  AF := 0x10 = (0x10 & (RAX ^ v18533 ^ 1))
  PF := ~low:1[let v18535 = RAX >> 4 ^ RAX in
    let v18535 = v18535 >> 2 ^ v18535 in
    v18535 >> 1 ^ v18535]
  SF := high:1[RAX]
  ZF := 0 = RAX
}
```

Used in BAP
https://github.com/BinaryAnalysisPlatform/bap

Representations

Many forms are possible...

- Conditional concurrent assignments
- Conditional sequential assignments
- Lisp
- k-Expressions
- C source code

```
(defun x86-add/adc/sub/sbb/or/and/xor/cmp/test-e-g
  (operation start-rip temp-rip prefixes
             rex-byte opcode modr/m sib x86)
  ;; Guards elided.
  (b* ((ctx 'x86-add/adc/sub/sbb/or/and/xor/cmp/test-E-G)
       (r/m (the (unsigned-byte 3) (mrm-r/m modr/m)))
       (mod (the (unsigned-byte 2) (mrm-mod  modr/m)))
       (reg (the (unsigned-byte 3) (mrm-reg  modr/m)))
       (lock? (eql #.*lock*
                   (prefixes-slice :group-1-prefix prefixes)))
       ((when (and lock? (eql operation #.*OP-CMP*)))
        ;; CMP does not allow a LOCK prefix.
        (!!ms-fresh :lock-prefix prefixes))

       (p2 (prefixes-slice :group-2-prefix prefixes))
       (byte-operand? (eql 0 (the (unsigned-byte 1)
                               (logand 1 opcode))))
       ((the (integer 1 8) operand-size)
        (select-operand-size byte-operand? rex-byte nil prefixes))
```

Used in the x86isa book for ACL2
http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____X86ISA

Representations

Many forms are possible...

- Conditional concurrent assignments
- Conditional sequential assignments
- Lisp
- k-Expressions
- C source code

Used in K Framework
http://www.kframework.org/index.php/Main_Page

```
// Autogenerated using stratification.
requires "x86-configuration.k"

module ADDQ-R64-R64
  imports X86-CONFIGURATION

  rule <k>
    execinstr (addq R1:R64, R2:R64,  .Operands) => .
  ...</k>
    <regstate>
RSMap:Map => updateMap(RSMap,
convToRegKeys(R2) |-> extractMInt( addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( mi(1,
RSMap))), 1, 65)

"CF" |-> extractMInt( addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParent
1)

"PF" |-> (#ifMInt (notBool (((((((eqMInt( extractMInt( addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), co
0), getParentValue(R2, RSMap))), 64, 65), mi(1, 1)) xorBool eqMInt( extractMInt( addMInt( concatenateMInt( mi(1, 0), getPa
concatenateMInt( mi(1, 0), getParentValue(R2, RSMap))), 63, 64), mi(1, 1))) xorBool eqMInt( extractMInt( addMInt( concate
getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParentValue(R2, RSMap))), 62, 63), mi(1, 1))) xorBool eqMInt( e
concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParentValue(R2, RSMap))), 61, 62), m
eqMInt( extractMInt( addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParent
61), mi(1, 1))) xorBool eqMInt( extractMInt( addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateM
getParentValue(R2, RSMap))), 59, 60), mi(1, 1))) xorBool eqMInt( extractMInt( addMInt( concatenateMInt( mi(1, 0), getPare
concatenateMInt( mi(1, 0), getParentValue(R2, RSMap))), 58, 59), mi(1, 1))) xorBool eqMInt( extractMInt( addMInt( concate
getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParentValue(R2, RSMap))), 57, 58), mi(1, 1)))) #then mi(1, 1) #

"AF" |-> xorMInt( xorMInt( extractMInt( getParentValue(R1, RSMap), 59, 60), extractMInt( getParentValue(R2, RSMap), 59, 6
addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParentValue(R2, RSMap))), 6

"ZF" |-> (#ifMInt eqMInt( extractMInt( addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( m
getParentValue(R2, RSMap))), 1, 65), mi(64, 0)) #then mi(1, 1) #else mi(1, 0) #fi)

"SF" |-> extractMInt( addMInt( concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParen
2)

"OF" |-> (#ifMInt ((eqMInt( extractMInt( getParentValue(R1, RSMap), 0, 1), mi(1, 1)) ==Bool eqMInt( extractMInt( getParen
1), mi(1, 1))) andBool (notBool (eqMInt( extractMInt( getParentValue(R1, RSMap), 0, 1), mi(1, 1)) ==Bool eqMInt( extractM
concatenateMInt( mi(1, 0), getParentValue(R1, RSMap)), concatenateMInt( mi(1, 0), getParentValue(R2, RSMap))), 1, 2), mi(
1) #else mi(1, 0) #fi)
)

    </regstate>

endmodule

module ADDQ-R64-R64-SEMANTICS
  imports ADDQ-R64-R64
endmodule
```

Representations

Many forms are possible…

- Conditional concurrent assignments
- Conditional sequential assignments
- Lisp
- k-Expressions
- C source code

```
SgAsmX86Instruction *
DisassemblerX86::decodeGroup1(SgAsmExpression* imm)
{
    switch (regField) {
        case 0: return makeInstruction(x86_add, "add", modrm, imm);
        case 1: return makeInstruction(x86_or, "or", modrm, imm);
        case 2: return makeInstruction(x86_adc, "adc", modrm, imm);
        case 3: return makeInstruction(x86_sbb, "sbb", modrm, imm);
        case 4: return makeInstruction(x86_and, "and", modrm, imm);
        case 5: return makeInstruction(x86_sub, "sub", modrm, imm);
        case 6: return makeInstruction(x86_xor, "xor", modrm, imm);
        case 7: return makeInstruction(x86_cmp, "cmp", modrm, imm);
        default: ASSERT_not_reachable("invalid reg field: " + StringUtility::numberToString(regField));
    }
    /* avoid MSCV warning by adding return stmt */
    return NULL;
}
```

Used in ROSE
http://rosecompiler.org/

Example Semantics: adc

```
// DWORD version.
adc(OP1:DWORD, OP2:DWORD) :=
  [true ->
    [ OP1 = OP1 + OP2 + bit(intel_CF)
    : intel_ZF = ((OP1 + OP2 + bit(intel_CF)) == 0)
    : intel_SF = ((signed(OP1 + OP2 + bit(intel_CF)) < 0))
    : intel_PF = is_even_parity_lowbyte((OP1 + OP2 + bit(intel_CF)))
    : intel_CF = carry_flag_adc_32(OP1, OP2, intel_CF)
    : intel_OF = overflow_flag_adc_32(OP1, OP2, intel_CF)
    : intel_AF = auxiliary_carry_flag_adc(OP1, OP2, intel_CF)
    ]
  ];
```

Example Semantics: inc

```
// DWORD version.
inc(OP1:DWORD) :=
  [true ->
    [ OP1 = OP1 + 1
    : intel_ZF = (unsigned_32(OP1) == (2**32 - 1))
    : intel_SF = (signed(OP1+1) < 0)
    : intel_PF = is_even_parity_lowbyte(OP1+1)
    : intel_OF = (unsigned_32(OP1) == (2**31 - 1))
    : intel_AF = ((unsigned_32(OP1) % 2**4) == ((2**4) - 1))
    ]
  ];
```

Example Semantics: add vs inc

```
// DWORD version.
add(eax, 1) :=
  [true ->
    [ eax = eax + 1
    : intel_ZF = (unsigned_32(eax) == (2**32 - 1))
    : intel_SF = ((signed(eax + 1) < 0))
    : intel_PF = is_even_parity_lowbyte(eax + 1)
    : intel_CF = carry_flag_add_32(eax, 1)
    : intel_OF = (unsigned_32(eax) == (2**31-1)
    : intel_AF = ((unsigned_32(eax) % 2**4) == ((2**4) -
1))
    ]
  ];
```

```
// DWORD version.
inc(eax) :=
  [true ->
    [ eax = eax + 1
    : intel_ZF = (unsigned_32(eax) == (2**32 - 1))
    : intel_SF = (signed(eax + 1) < 0)
    : intel_PF = is_even_parity_lowbyte(eax + 1)

    : intel_OF = (unsigned_32(eax) == (2**31 - 1))
    : intel_AF = ((unsigned_32(eax) % 2**4) == ((2**4) -
1))
    ]
  ];
```

Example Semantics: add vs inc

```
// DWORD version.
add(eax, 1) :=
  [true ->
    [ eax = eax + 1
    : intel_ZF = (unsigned_32(eax) == (2**32 - 1))
    : intel_SF = ((signed(eax + 1) < 0))
    : intel_PF = is_even_parity_lowbyte(eax + 1)
    : intel_CF = carry_flag_add_32(eax, 1)
    : intel_OF = (unsigned_32(eax) == (2**31-1)
    : intel_AF = ((unsigned_32(eax) % 2**4) == ((2**4) -
1))
    ]
  ];
```

```
// DWORD version.
inc(eax) :=
  [true ->
    [ eax = eax + 1
    : intel_ZF = (unsigned_32(eax) == (2**32 - 1))
    : intel_SF = (signed(eax + 1) < 0)
    : intel_PF = is_even_parity_lowbyte(eax + 1)

    : intel_OF = (unsigned_32(eax) == (2**31 - 1))
    : intel_AF = ((unsigned_32(eax) % 2**4) == ((2**4) -
1))
    ]
  ];
```

Adequate for one purpose is not necessarily adequate for all purposes.

It gets worse.

CPU state (only showing rax)

```
rax = 0xa3d903d69abcfdb0
                    ----
                     %ax
```

```
addw $0x1, %ax
```

CPU state (only showing rax)

```
rax = 0xa3d903d69abcfdb1

           fdb0+1=fdb1
```

CPU state (only showing rax)

```
rax = 0xa3d903d69abcfdb0
               ---------
                  %eax
```

```
addl $0x1, %eax
```

CPU state (only showing rax)

```
rax = 0x000000009abcfdb1

     9abcfdb0+1=9abcfdb1
```

https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/

Lots of instructions!

Measure	Count	Comment
AT&T mnemonic (e.g., `addl`)	1,279	Counts the number of unique mnemonics in AT&T syntax.
Intel mnemonic (e.g., `add`)	981	This is a rough estimate of the number of different kinds of operations the x86 instruction set can perform, ignoring the operand type and size. There are various caveats as described earlier, such as some operations having multiple different mnemonics.
Mnemonic and operand types (e.g., `add_r32_imm32`)	3,683	Distinguishes instructions in every way possible and is thus a sort of upper bound on the number of instructions. If you are looking for a very fine-grained notion of instruction, this is a good measure.
Mnemonic and operand width (e.g., `add_32_32`)	2,034	This is an estimate of the number of different kinds of instructions, if an operation on 8 bits is considered to be different from the same operation on 16 bits (because sometimes, these actually have different semantics, as shown with the zeroing of the upper 32 bits for the `addl` instruction). Does not distinguish instructions that operate on registers vs. constants vs. memory locations.
Every valid bit sequence (e.g., `0x83`, `0xC0`, `0x01`)	?	Counts every bit seqeuence that makes a valid x86 instruction. Unfortunately there are too many to count.

https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/

one-byte opcodes index:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
```

two-byte opcodes (0F..) index:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
```

pf	0F	po	so	o	proc	st	m	rl	x	mnemonic	op1	op2	op3	op4	iext	tested f	modif f	def f	undef f	f values	description, notes
		00		r					L	ADD	**r/m8**	r8					o..szapc	o..szapc			Add
		01		r					L	ADD	**r/m16/32/64**	r16/32/64					o..szapc	o..szapc			Add
		02		r						ADD	**r8**	r/m8					o..szapc	o..szapc			Add
		03		r						ADD	**r16/32/64**	r/m16/32/64					o..szapc	o..szapc			Add
		04								ADD	**AL**	imm8					o..szapc	o..szapc			Add
		05								ADD	**rAX**	imm16/32					o..szapc	o..szapc			Add
		06					E			*invalid*											Invalid Instruction in 64-Bit Mode
		07					E			*invalid*											Invalid Instruction in 64-Bit Mode
		08		r					L	OR	**r/m8**	r8					o..szapc	o..sz.pc	.....a..	o......c	Logical Inclusive OR
		09		r					L	OR	**r/m16/32/64**	r16/32/64					o..szapc	o..sz.pc	.....a..	o......c	Logical Inclusive OR
		0A		r						OR	**r8**	r/m8					o..szapc	o..sz.pc	.....a..	o......c	Logical Inclusive OR
		0B		r						OR	**r16/32/64**	r/m16/32/64					o..szapc	o..sz.pc	.....a..	o......c	Logical Inclusive OR
		0C								OR	**AL**	imm8					o..szapc	o..sz.pc	.....a..	o......c	Logical Inclusive OR
		0D								OR	**rAX**	imm16/32					o..szapc	o..sz.pc	.....a..	o......c	Logical Inclusive OR
		0E					E			*invalid*											Invalid Instruction in 64-Bit Mode
		0F								*Two-byte Instructions*											
		10		r					L	ADC	**r/m8**	r8				c	o..szapc	o..szapc			Add with Carry
		11		r					L	ADC	**r/m16/32/64**	r16/32/64				c	o..szapc	o..szapc			Add with Carry
		12		r						ADC	**r8**	r/m8				c	o..szapc	o..szapc			Add with Carry
		13		r						ADC	**r16/32/64**	r/m16/32/64				c	o..szapc	o..szapc			Add with Carry
		14								ADC	**AL**	imm8				c	o..szapc	o..szapc			Add with Carry
		15								ADC	**rAX**	imm16/32				c	o..szapc	o..szapc			Add with Carry
		16					E			*invalid*											Invalid Instruction in 64-Bit Mode
		17					E			*invalid*											Invalid Instruction in 64-Bit Mode
		18		r					L	SBB	**r/m8**	r8				c	o..szapc	o..szapc			Integer Subtraction with Borrow
		19		r					L	SBB	**r/m16/32/64**	r16/32/64				c	o..szapc	o..szapc			Integer Subtraction with Borrow

Source: http://ref.x86asm.net/

Next Time:
Slicing on Semantics