CS142B Language Processor Construction

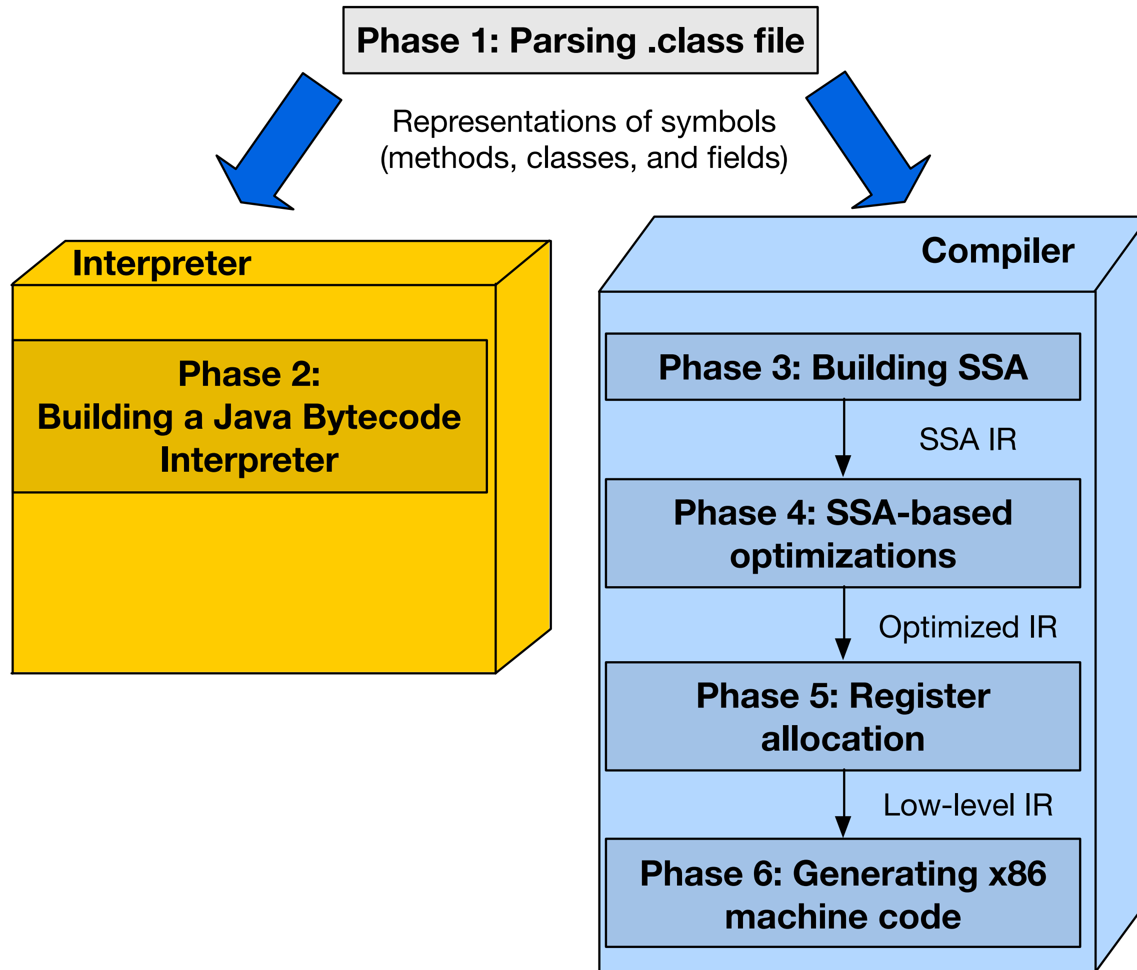
Code Generation

Yeoul Na
UCI

May 28, 2019

(updated May 28, 2019)

Project Overview



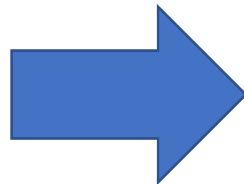
IR to Machine Code

```
BB0: (Entry)
MOV R1, #0
MOV R2, #0

BB4:
CMP R1, #5
BR_GE BB19

BB9:
ADD R2, R1
INC R1
JMP BB4

BB19:
PUSH R2
CALL “printInt”
RETURN
```



IR to Machine Code

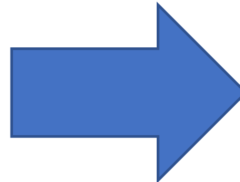
- Computers understand numbers

```
BB0: (Entry)
MOV R1, #0
MOV R2, #0

BB4:
CMP R1, #5
BR_GE BB19

BB9:
ADD R2, R1
INC R1
JMP BB4

BB19:
PUSH R2
CALL “printInt”
RETURN
```



011010…
…

Machine Code Generation

- Machine Code Generation? One to one translation from IR to binary encoding

Machine Code Generation

- Machine Code Generation? One to one translation from IR to binary encoding

- Instruction Set Architecture
  - Instruction format: ADD rdi,0x38
  - Binary encoding (hex): 48 83 C7 38
- Calling Convention
  - How function parameters are passed (registers or stack)
  - Which registers the callee must preserve for the caller
    - Caller vs callee-saved registers

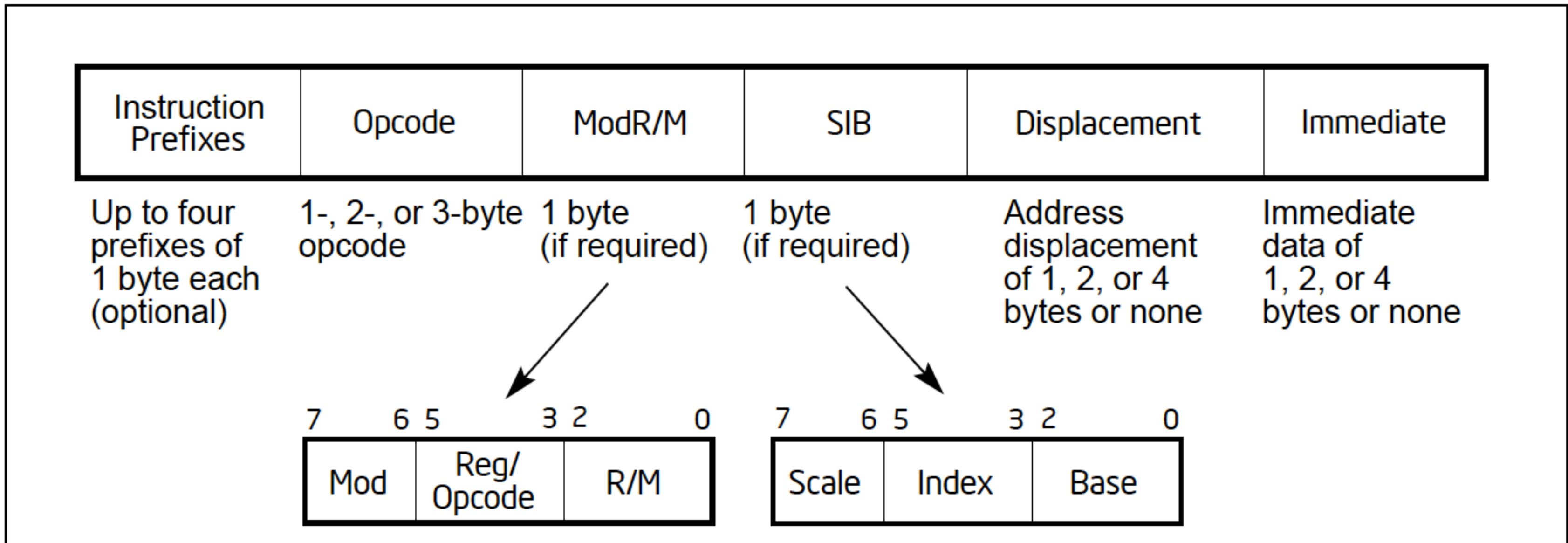
Instruction Set Architecture (ISA)

- An abstract model of a computer
- Serves as the interface between software and hardware
- ISA defines
  - Available instructions: ADD, SUB, ...
  - Operand types: Byte, Word, Double-word, ...
  - Registers: EAX, EBX, ECX, ...
  - Addressing modes
  - Binary encoding of instructions
  - Etc.

x86 Instructions We Support

- add, sub, mov, call, push, pop, ret, jmp, jcc, cmp

X86 Instruction Format



The Intel 64 and IA-32 Architectures Software Developer's Manuals' "CHAPTER 2 INSTRUCTION FORMAT"

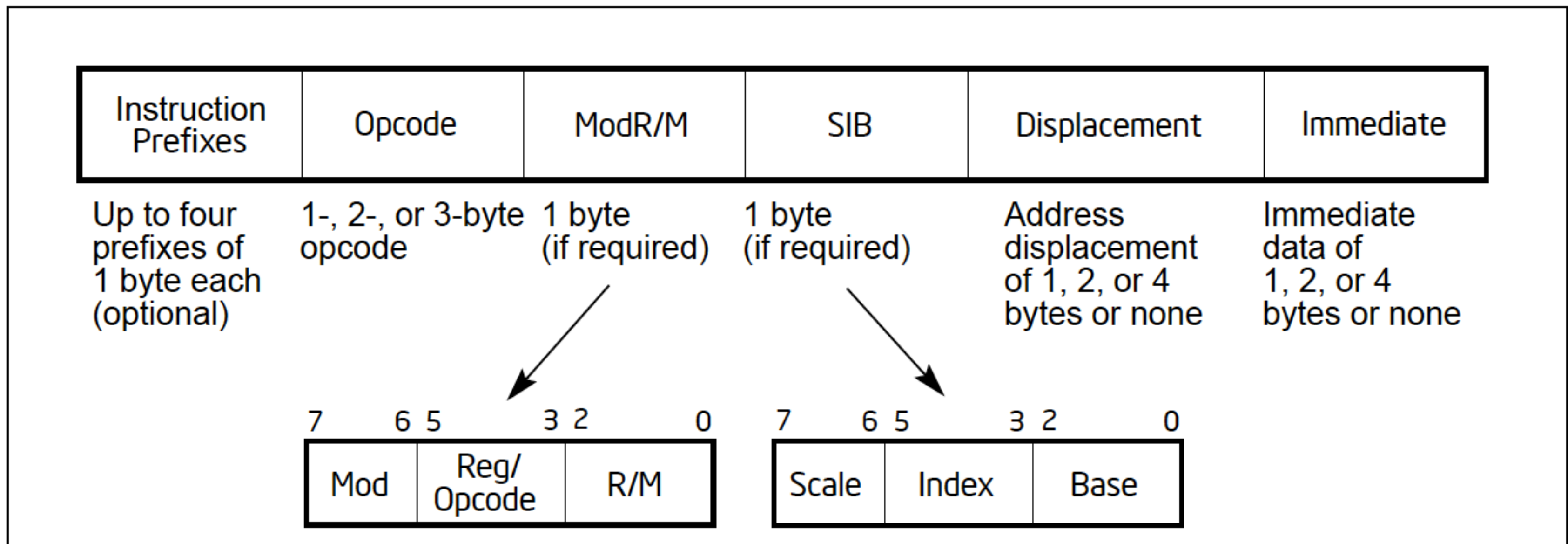
Instruction Prefix

- REX Prefixes
  - Specify GPRs and SSE registers
  - Specify 64-bit operand size
  - Specify extended control registers
- Ex) 64-bit operand : REX.W => 0100 1000 (0x48)

Table 2-4. REX Prefix Fields [BITS: 0100WRXB]

Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D
		1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field

ModR/M Byte



- mod : define addressing modes
- reg/opcode : specify either a register number or three more bits of opcode information
- r/m : specify a register as an operand or can be combined with the mod field to encode an addressing mode

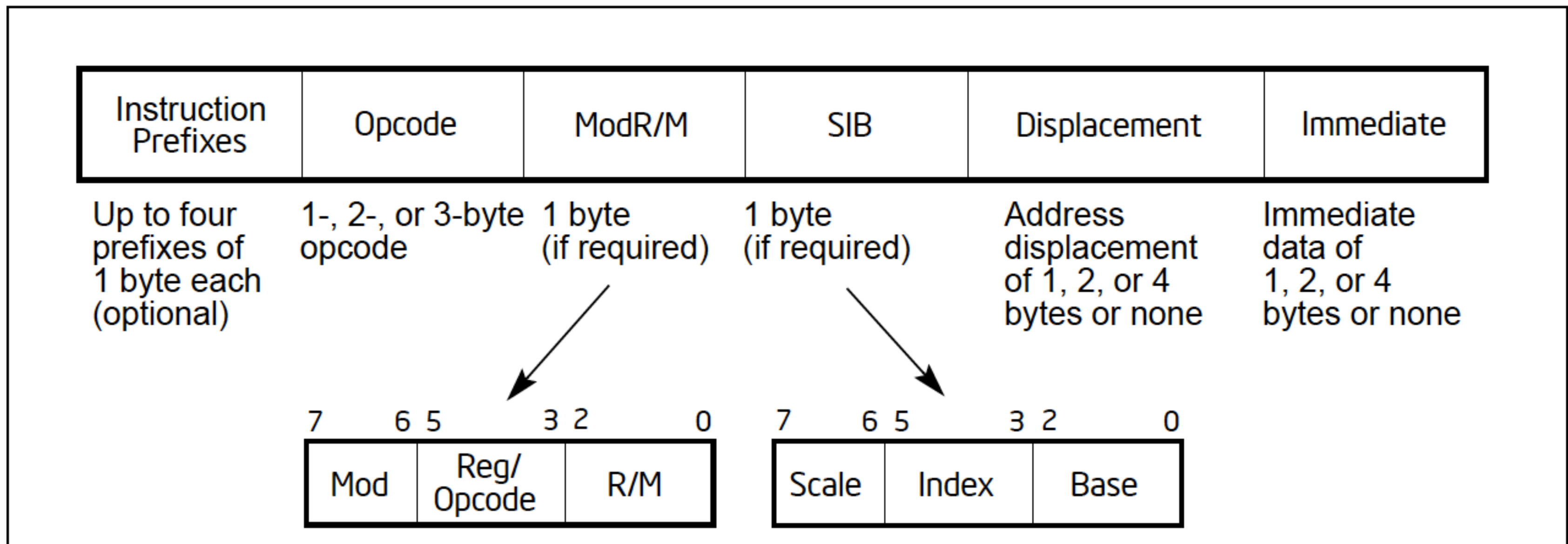
Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Displacement and Immediate

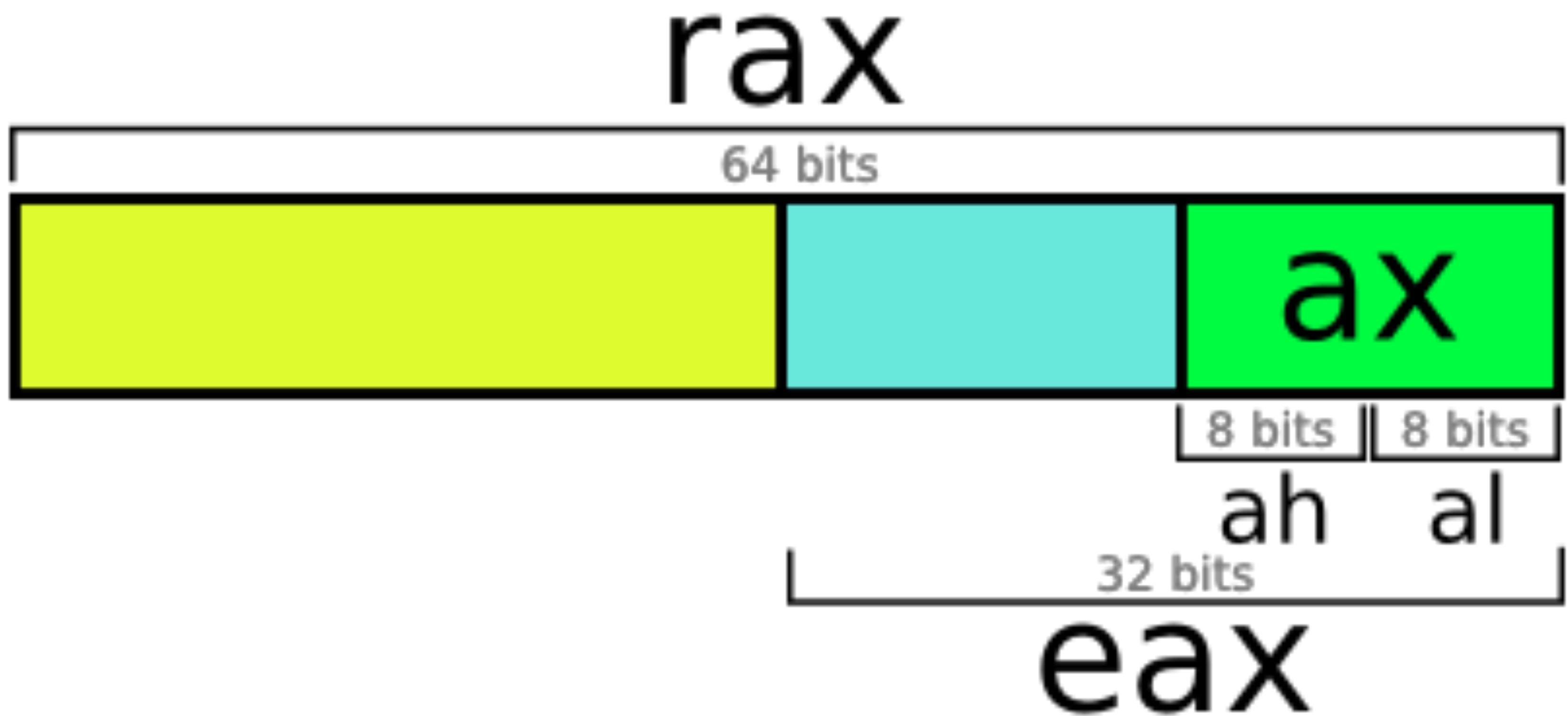


- Displacement: [reg + displacement]
  - (mov [rbp+8], rax)
- Immediate: an immediate operand

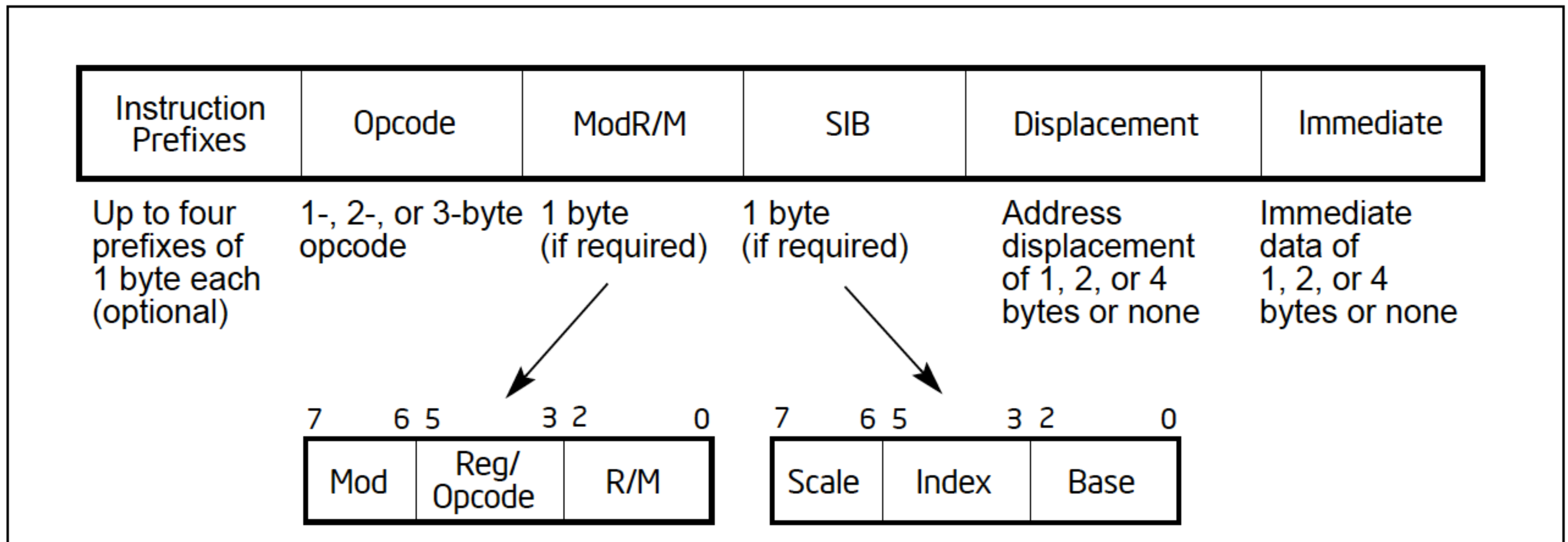
ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.
REX.W + 05 id	ADD RAX, imm32	I	Valid	N.E.	Add imm32 sign-extended to 64-bits to RAX.
80 /0 ib	ADD r/m8, imm8	MI	Valid	Valid	Add imm8 to r/m8.
REX + 80 /0 ib	ADD r/m8*, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m8.
81 /0 iw	ADD r/m16, imm16	MI	Valid	Valid	Add imm16 to r/m16.
81 /0 id	ADD r/m32, imm32	MI	Valid	Valid	Add imm32 to r/m32.
REX.W + 81 /0 id	ADD r/m64, imm32	MI	Valid	N.E.	Add imm32 sign-extended to 64-bits to r/m64.
83 /0 ib	ADD r/m16, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m16.
83 /0 ib	ADD r/m32, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m32.
REX.W + 83 /0 ib	ADD r/m64, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m64.
00 /r	ADD r/m8, r8	MR	Valid	Valid	Add r8 to r/m8.
REX + 00 /r	ADD r/m8*, r8*	MR	Valid	N.E.	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8, r/m8	RM	Valid	Valid	Add r/m8 to r8.
REX + 02 /r	ADD r8*, r/m8*	RM	Valid	N.E.	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

X86 Register Widths



Example : ADD rdi, 0x38



Opcode	Instruction
REX.W + 83 /0 ib	ADD r/m64, imm8

- Instruction format: ADD rdi,0x38 => ModR/M: 11000111 (0xc7)
- Binary encoding (hex): 48 83 C7 38

Exercise!

Opcode	Instruction
E8 cd	CALL rel32
REX.W + 01 /r	ADD r/m64, r64
REX.W + 89 /r	MOV r/m64, r64
REX.W + 83 /0 ib	ADD r/m64, imm8

REX.W == 0x48

1) CALL 0x34:

2) ADD rax, rdi:

3) MOV [rbp – 8], rbx:

4) ADD rax, 0x12:

Exercise!

Opcode	Instruction
E8 cd	CALL rel32
REX.W + 01 /r	ADD r/m64, r64
REX.W + 89 /r	MOV r/m64, r64
REX.W + 83 /0 ib	ADD r/m64, imm8

REX.W == 0x48

1) CALL 0x34: E8 34 00 00 00

2) ADD rax, rdi: 48 01 F8

3) MOV [rbp – 8], rbx: 48 89 5D F8

4) ADD rax, 0x12: 48 83 C0 12

Calling Convention

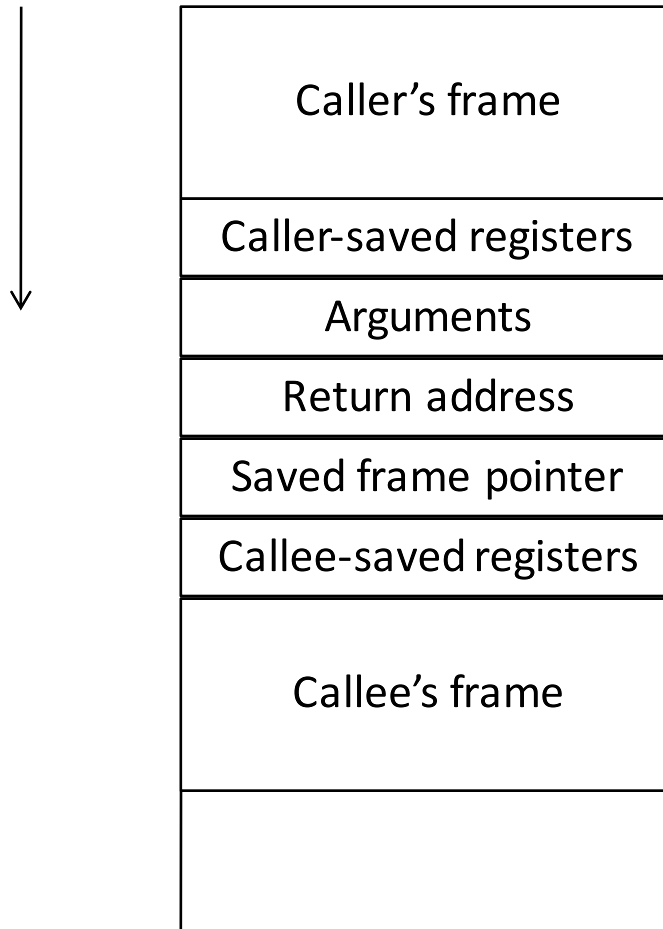
- The interface between caller / callee
    - How parameters are passed (registers or stack)
    - Which registers the callee must preserve for the caller
        - Caller vs callee-saved registers
    - Determines the stack layout

Passing Arguments

x86-64	Microsoft x64 calling convention[14]	Windows (Microsoft Visual C++, GCC, Intel C++ Compiler, Delphi), UEFI	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3
	vectorcall	Windows (Microsoft Visual C++)	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3 + XMM0-XMM5/YMM0-YMM5
	System V AMD64 ABI[19]	Solaris, Linux, BSD, OS X (GCC, Intel C++ Compiler)	RDI, RSI, RDX, RCX, R8, R9, XMM0–7

- https://en.wikipedia.org/wiki/X86_calling_conventions

X86 Stack Layout



IR to Binary Translation

- Input: CFG, Instruction sequence, Value to Reg map
- Output: byte stream (in byte buffer)
- Sequentially visit Instructions
  - An instruction contains opcode and operands
  - You can tell what instruction it is and what machine registers you should use
- E.g.: if (auto BI = dynamic_cast<BinaryInst*>(I))
  - If (BI->getOpcode() == OP_ADD)
    - emitByte(0x48)
    - emitByte(0x83) …

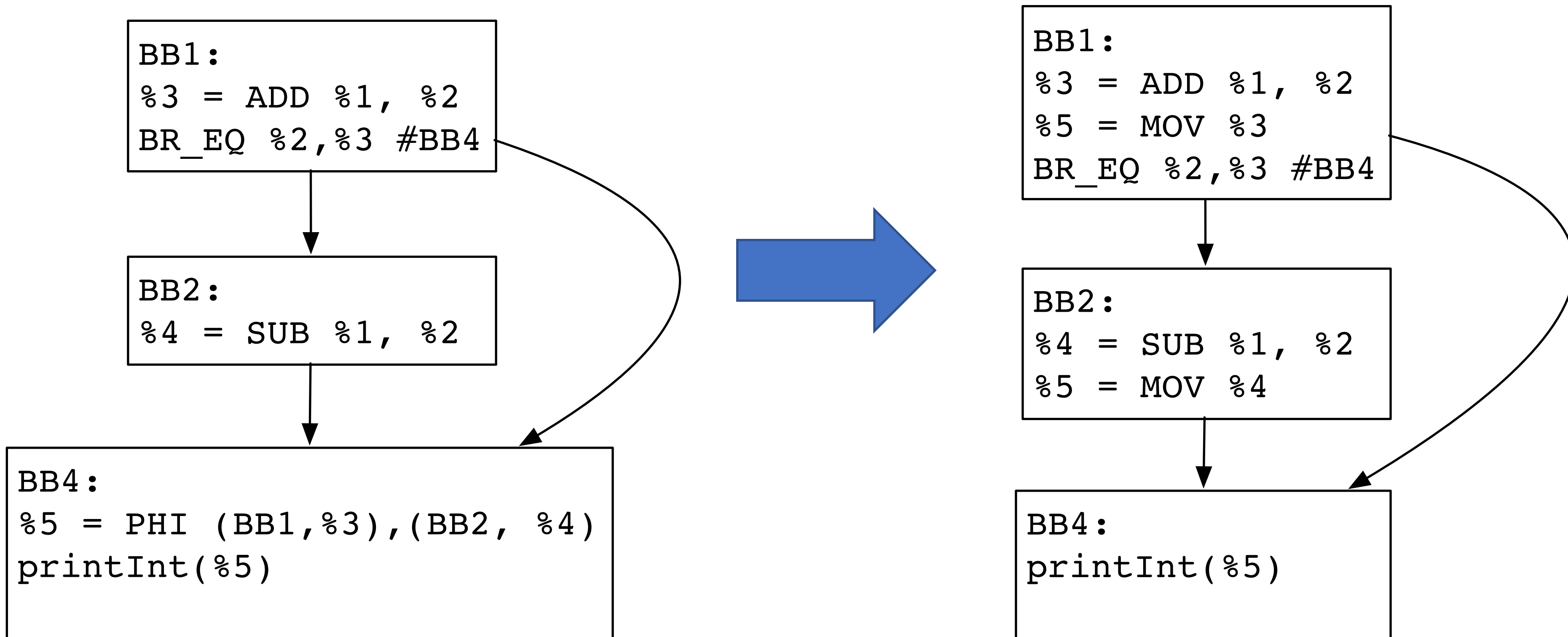
Handling Binary Instruction

- IR: ADD R1, R2, R3 (three-address code)
- X86: ADD R1, R3 (two-address code)
  - R1 is read and then written in the same instruction
- Insert move instruction before ADD instructions!

ADD R1, R2, R3 ⇒ **MOV R1, R2**
ADD R1, R3

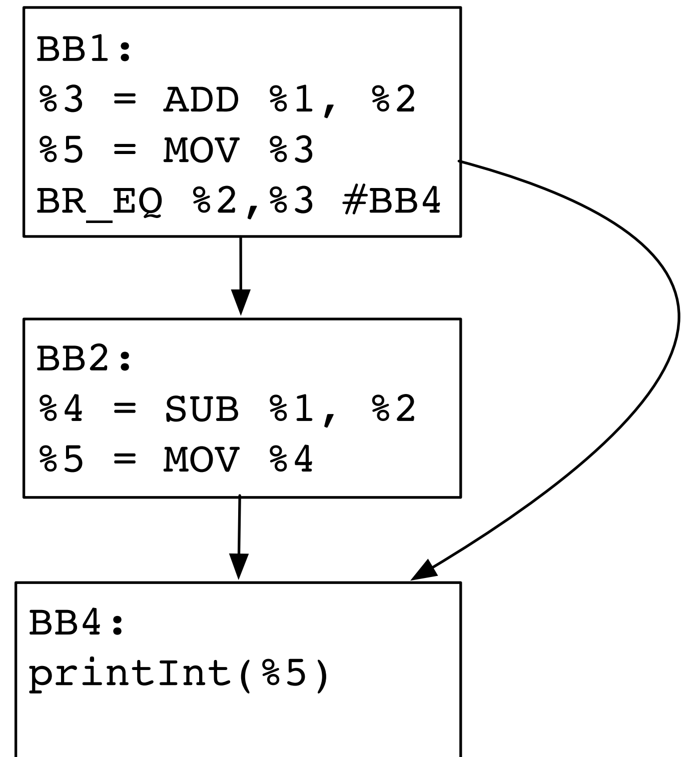
Handling Phi function

- At the end of Basic Block, check Phi functions in successors. Insert MOVs to the Phi destination.



Handling Branch / Jump

- You may not know yet the actual byte offset of the target instruction
  - Maintain relocation table!
- Keep two data structures
  - Inst (leader) to Offset map
  - Relocation Table
    - Instruction to offsets that need to be fixed!



Handling function calls

- Store caller-saved registers before calling
- Prepare arguments for the call
- Call the target function
- Recover caller-saved registers after calling

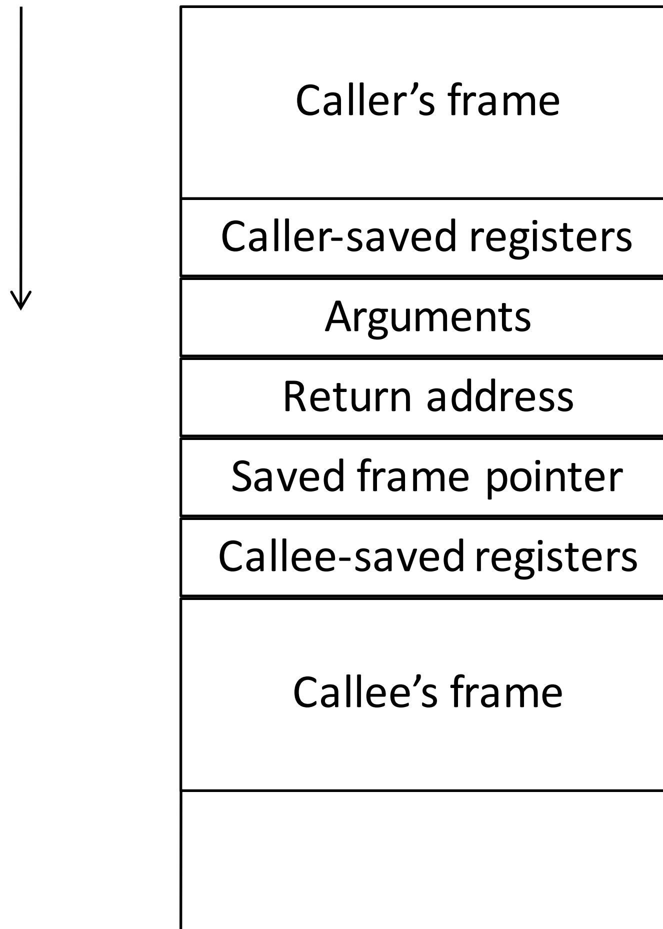
x86-64 calling conventions

- Linux (System V AMD64 ABI)
  - Caller-saved: RAX, RCX, RDX, RDI, RSI
  - Callee-saved: RBX
- Microsoft Windows
  - Caller-saved: RAX, RCX, RDX
  - Callee-saved: RBX, RDI, RSI

Handling call to PrintInt(int)

- Store caller-saved registers before calling
- Prepare arguments for the call
  - Move the argument to RDI (Linux) / RCX (Windows)
- Call the target function
  - Define a function that prints integer in your compiler
  - Get the address of PrintInt : &PrintInt
  - Calculate the relative address from the end of the instruction to the target function
- Recover caller-saved registers after calling

X86 Stack Layout



JIT Code Cache

- Where the executable bytes are emitted
- You will call this region!
- Memory map a Writable & Executable region
- Linux / OS X: mmap (sys/mman.h) or mprotect
  - E.g.
    ```
    JITCode = mmap(nullptr, (4 << 10), PROT_EXEC | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
    ((void(*)())JITCode();
    ```
- Windows: HeapCreate (memoryapi.h)
  - E.g.
    ```
    Handle = HeapCreate(HEAP_CREATE_ENABLE_EXECUTE, …)
    JITCode = HeapAlloc(Handle, …)
    ```

Function Prologue / Epilogue

- Function prologue (start of the function)
  - Push caller's frame pointer
  - Prepare the caller's stack frame (decrement)
  - Save callee-saved registers
- Function epilogue (right before return instructions)
  - Recover callee-saved registers
  - Recover the original stack pointer
  - Pop caller's frame pointer

Prologue / Epilogue for Linux

- Prologue

```
55              pushq  %rbp
48 89 e5        movq   %rsp, %rbp
48 83 ec 30     subq   $0x30, %rsp
48 89 5d f8     movq   %rbx, -0x8(%rbp)
```

- Epilogue

```
48 8b 5d f8     movq   -0x8(%rbp), %rbx
48 83 c4 30     addq   $0x30, %rsp
5d              popq   %rbp
```

8* StackSize;
StackSize =
#Spills + #Callee-saved registers
(if there's a function call) + #Caller-saved registers

Step by Step

- Allocate executable memory region
- Emit function prologue
- Sequentially iterate over instructions
  - Handle each instruction based on its type
    - Special care: BinaryInst, PhiFunction, CallInst
    - Emit right bytes for the instruction and operands as specified in ISA
    - Emit epilogue right before each return instruction

Tips - objdump

- objdump –d obj.o > obj.asm

```
Disassembly of section .text:

00000001000009f0 <_main>:
   1000009f0:   55                      push   %rbp
   1000009f1:   48 89 e5                mov    %rsp,%rbp
   1000009f4:   48 83 ec 20             sub    $0x20,%rsp
   1000009f8:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   1000009ff:   89 7d f8                mov    %edi,-0x8(%rbp)
   100000a02:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
   100000a06:   48 8b 75 f0             mov    -0x10(%rbp),%rsi
   100000a0a:   48 8b 76 08             mov    0x8(%rsi),%rsi
   100000a0e:   48 8d 7d e0             lea    -0x20(%rbp),%rdi
   100000a12:   e8 19 00 00 00          callq  100000a30 <__ZN3JVMC1EPKc>
   100000a17:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   100000a1e:   48 8d 7d e0             lea    -0x20(%rbp),%rdi
   100000a22:   e8 39 00 00 00          callq  100000a60 <__ZN3JVMD1Ev>
   100000a27:   8b 45 fc                mov    -0x4(%rbp),%eax
   100000a2a:   48 83 c4 20             add    $0x20,%rsp
   100000a2e:   5d                      pop    %rbp
   100000a2f:   c3                      retq
```

Tips – gdb, lldb

- Disassemble your JIT code (jitptr) in memory
  - (gdb) disas **jitptr**, +**length**
  - (lldb) disas -s **jitptr** -b -c **length**
  - Replace jitptr and length with yours
- E.g.

```
(gdb) disas 0x32c4,+32
Dump of assembler code from 0x32c4 to 0x32e4:
0x32c4 <main+204>: addil 0,dp
0x32c8 <main+208>: ldw 0x22c(sr0,r1),r26
0x32cc <main+212>: ldil 0x3000,r31
…
```

Disassembly syntax

- Intel vs AT&T syntax
  - AT&T : “sub $0x20,%rsp” (default in gdb, objdump)
  - Intel : “sub rsp, 0x20”
- Changing the setting
  - (gdb) disassembly-flavor intel
  - objdump –d –M intel obj.o > obj.asm