

Back-end

- Runtime model
- Code generation
- Register Allocation
- Code emission



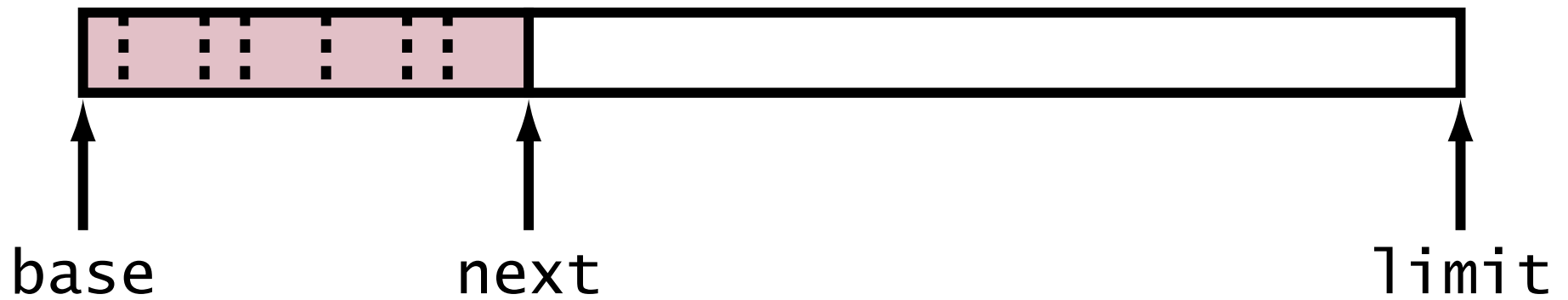
Back-end

- Generate executable code from FIR
- A process runs as part of a *runtime environment*
- Normal Java: simple runtime
 - *Stack allocation of frames*
- FJava
 - *Heap allocation of frames => garbage collection*
 - *(Heap allocation with malloc; free is nop)*
 - *Runtime checks*

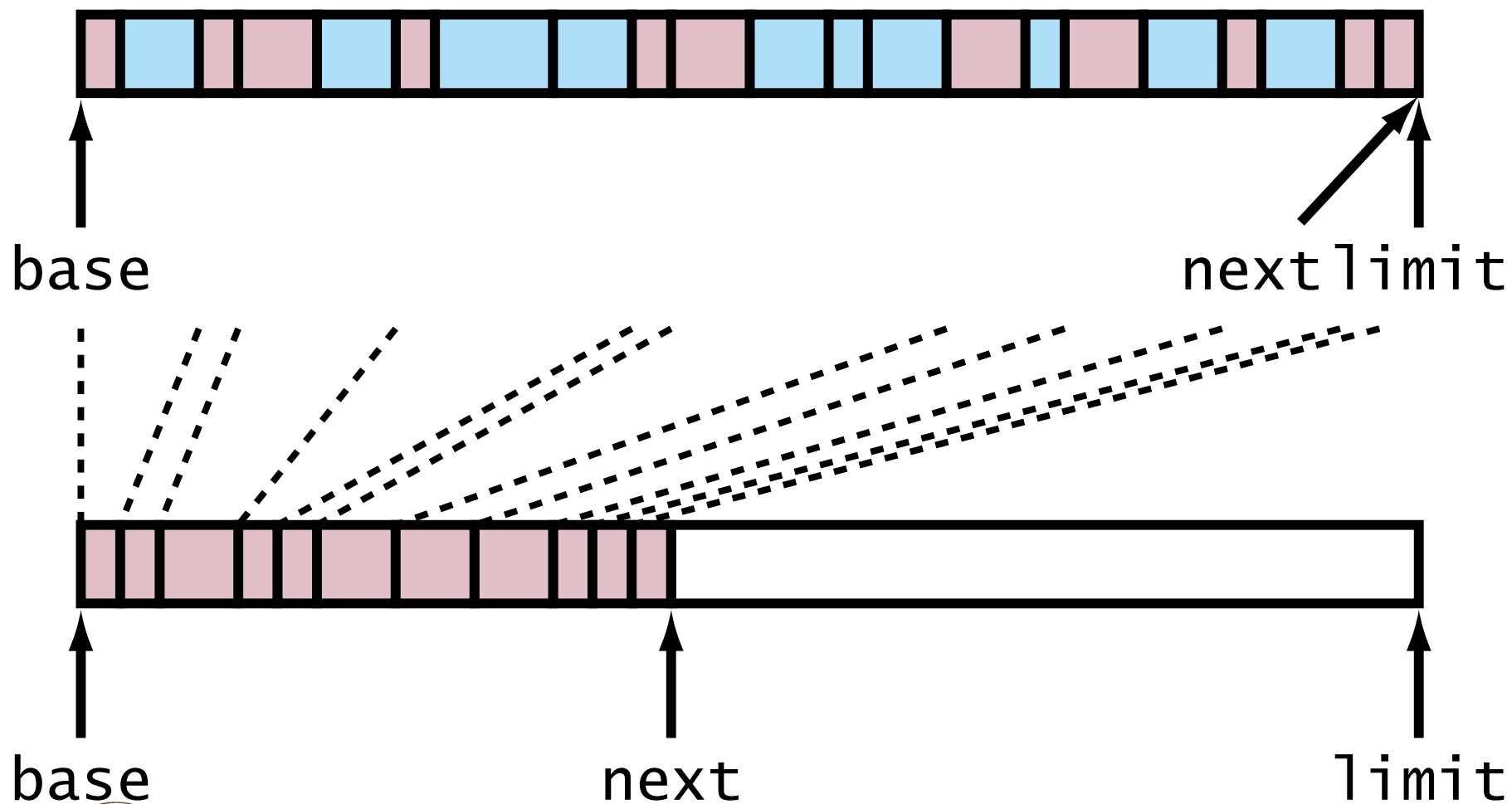


Brief overview of runtime

- Contiguous allocation



Heap compaction



Garbage Collection

- We will be using a simple copying collector
- We'll talk more about this after code generation

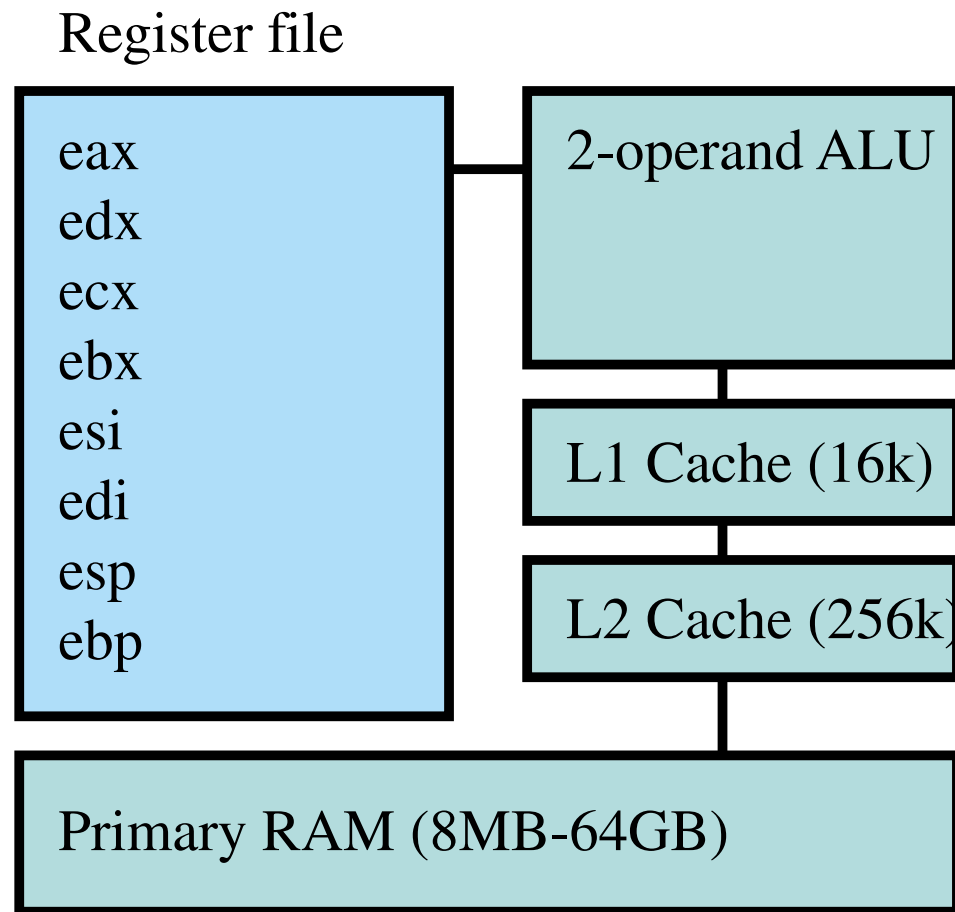


Stages in backend

- Translate FIR to x86 assembly
- Perform register allocation
- Emit code
- Run!



Oversimplified x86 architecture



2-operand instructions

```
// Factorial:  
// Arg in %ebx  
// Result in %eax  
// Destroys %edx  
mov    %eax,$1      // %eax <- 1  
  
fact:  
    cmp    %ebx,$0    // test %ebx == 0  
    jmp    z,break    // if so, exit  
    mul    %ebx       // %eax *= %ebx  
    dec    %ebx       // %ebx--  
    jmp    fact       // next iteration  
  
exit:
```



Notes

- Most instructions have a normal 2-operand form
 - *ADD op1,op2 means op1 += op2*
- Some instructions are really strange
 - *MUL op1 means (edx,eax) *= op1*
 - *SHL op1,op2 means op1 <<= op2, but op2 must be a constant or %cl*



x86 is a CISC architecture

- Lots of instructions, some very complex
 - *For example, looping constructs, string operations*
- We will use only a simple subset
- Most complex instructions are pretty slow
 - *Because compiler writers ignore the complex parts*



Addressing

- x86 supports complex addressing modes
 - *On a RISC, there is only 1 mode: $*(register+short_offset)$*
- x86 operands:
 - *Register r: value in register r*
 - *Immediate: numeric constants*
 - *Label: value at memory location with label l*
 - *MemReg r: $*r$*
 - *MemRegOff (r, off): $*(r+off)$*
 - *MemRegRegOffMul (r1, r2, off, mul): $*(r1 + r2 * mul + off)$*
- In two address instructions, usually one operand has a simple form
 - *Register, Immediate, Label*



Operands

type reg = symbol

type var = symbol

type label = symbol

type operand =

ImmediateNumber of int

| ImmediateLabel of label

| ImmediateCLabel of label

| Register of reg

| SpillRegister of int

| MemReg of reg

| MemRegOff of reg * int

| MemRegRegOffMul of reg * reg * int * int



X86 instruction set

- We'll use a simplified representation
- Note: we'll initially assume that there are an infinite number of registers
 - *Register v is valid for any variable v*
 - *Register allocation will take care of assignment to actual registers*



Copying

- MOV

type dst = operand

type src = operand

type inst =

(** Copy **)

MOV of dst * src

| MOVB of dst * src (** byte copy **)



Arithmetic

(Arithmetic *)*

| *NEG of dst*

| *NOT of dst*

| *ADD of dst * src*

| *LEA of dst * src*

| *SUB of dst * src*

| *MUL of dst*

| *IMUL of dst * src*

| *DIV of dst*

| *CDQ*

| *IDIV of dst*

| *AND of dst * src*

| *OR of dst * src*

| *XOR of dst * src*

| *SAR of dst * src*

| *SHL of dst * src*

| *SHR of dst * src*



Control flow

<i>(* Stack operations *)</i>	<i>type cc =</i>
<i>PUSH of src</i>	<i>LT</i>
<i>CALL of src</i>	<i>LE</i>
	<i>EQ</i>
<i>(* Branching *)</i>	<i>NEQ</i>
<i>TEST of src * src</i>	<i>GT</i>
<i>CMP of src * src</i>	<i>GE</i>
<i>JMP of label</i>	<i>ULT</i>
<i>JCC of cc * label</i>	<i>ULE</i>
<i>IJMP of src list * dst</i>	<i>UGT</i>
<i>SET of cc * dst</i>	<i>UGE</i>



Pseudo-operations

(Pseudo-code for memory management *)*
*| GC of int * src list*

(Comment pseudo-codes *)*
| CommentFIR of Fj_fir.exp



The “Frame”

- The X86Frame module defines useful architecture-specific parameters

(List of general-purpose registers *)*

val registers : var list

val registers_special : var list

(Get the standard argument list: pass the param count *)*

val get_stdargs : int -> var list

...

(Standard register names *)*

val eax : var

val ebx : var

val ecx : var



Utilities: Listbuf

- Just like a list, with constant-time append

Listbuf module:

type 'a t

val empty : 'a t

val add : 'a t -> 'a -> 'a t

val to_list : 'a t -> 'a list



Code generation: atoms

- Atoms become operands

```
let build_atom a =  
  match a with  
  | AtomUnit  
  | AtomNil ->  
    ImmediateNumber 0  
  | AtomBool b ->  
    ImmediateNumber (if b then 1 else 0)  
  | AtomChar c ->  
    ImmediateNumber (Char.code c)  
  | AtomInt i ->  
    ImmediateNumber i  
  | AtomFloat _ ->  
    raise (Failure "code_atom: floats not implemented")  
  | AtomVar v ->  
    Register v
```



Code generation: LetAtom

- LetAtom (v, ty, a, e) becomes a MOV

```
and build_atom_exp info blocks insts pos v a e =  
  let pos = string_pos "build_atom_exp" pos in  
  let insts = Listbuf.add insts (MOV (Register v, build_atom a)) in  
    build_exp info blocks insts e
```



Code generation: LetBinop

- Part I: construct operands

```
and build_binop_exp info blocks insts pos v op a1 a2 e =  
  let pos = string_pos "build_binop_exp" pos in  
  let a1 = build_atom a1 in  
  let a2 = build_atom a2 in  
  let dst = Register v in
```

...



LetBinop: Add

- Two-operand arithmetic
 - *Load dst with first operand*
 - *Operate on it with second operand*

match op with

AddIntOp ->

let insts = Listbuf.add insts (MOV (dst, a1)) in

let insts = Listbuf.add insts (ADD (dst, a2)) in
insts



LetBinop: DIV

- Bogus architecture strikes again

| DivIntOp ->

```
let tmp = new_symbol_string "op2" in  
let insts = Listbuf.add insts (MOV (Register eax, a1)) in  
let insts = Listbuf.add insts CDQ in  
let insts = Listbuf.add insts (MOV (Register tmp, a2)) in  
let insts = Listbuf.add insts (IDIV (Register tmp)) in  
let insts = Listbuf.add insts (MOV (dst, Register eax)) in  
insts
```



LetSubscript (v, ty, a1, a2, e)

- Check that a1 is not nil
- Check that a2 is in bounds
- Get the value



LetSubscript, nil-check

let insts = Listbuf.add insts (MOV (Register v_array, a1)) **in**

let insts = Listbuf.add insts (CMP (Register v_array, ImmediateNumber 0)) **in**

let insts = Listbuf.add insts (JCC (EQ, seg_fault_label)) **in**



LetSubscript, bounds-check

```
let insts = Listbuf.add insts (MOV (Register size, MemRegOff (v_array, header_off))) in  
let insts = Listbuf.add insts (SHR (Register size, ImmediateNumber size_shift)) in  
let insts = Listbuf.add insts (AND (Register size, ImmediateNumber size_mask)) in  
let insts = Listbuf.add insts (MOV (Register v_index, a2)) in  
let insts = Listbuf.add insts (CMP (Register size, Register v_index)) in  
let insts = Listbuf.add insts (JCC (ULE, seg_fault_label)) in
```



LetSubscript, productive work

```
let insts = Listbuf.add insts (MOV (Register v, MemRegRegOffMul (v_array, v_index, 0, 4))) in  
  build_exp info blocks insts e
```

