Generating Pentium Code

x86 Quick Overview

- Registers:
  - General purpose 32bit: eax, ebx, ecx, edx, esi, edi
    - Also 16-bit: ax, bx, etc., and 8-bit: al, ah, bl, bh, etc.
  - Stack registers:
    - esp: stack pointer (like SaM SP, points to topmost stack address)
    - ebp: frame base pointer (like SaM FBR)
- Instructions:
  - Arithmetic: add, sub, inc, mod, idiv, imul, etc.
  - Logic: and, or, not, xor
  - Comparison: cmp, test
  - Control flow: jmp, jcc, jecz
  - Function calls: call, ret
  - Data movement: mov (many variants)
  - Stack manipulations: push, pop
  - Other: lea

Note on register names

Registers are general-purpose: can be used for anything programmer wants

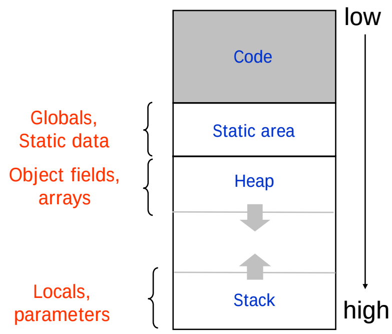
Historically, the registers were intended to be used as shown below, hence their odd names:

- AX/EAX/RAX: accumulator
- BX/EBX/RBX: base
- CX/ECX/RCX: counter
- DX/EDX/RDX: data/general
- SI/ESI/RSI: "source index" for string operations.
- DI/EDI/RDI: "destination index" for string operations.
- SP/ESP/RSP: stack pointer for top address of the stack.
- BP/EBP/RBP: stack base pointer for holding the address of the current stack frame.
- IP/EIP/RIP: instruction pointer. Holds the current instruction address.

Instruction set

- SaM: zero-address instruction set
  - instructions popped operands from stack and pushed result on stack
- X86 instruction set: two-address instruction set
  - Op a, b
    - a,b specify the two operands
    - result of operation is stored in b
    - a,b: registers or memory address
    - at most one operand can be in memory
    - memory addresses can be specified as offset from ebp (or other registers)
      - SaM: PUSHOFF 8 ➔ x86: pushl 8(%ebp)
      - more generally, address can be specified as disp(base,offset,scale)
  - Examples:
    - addl $3, %eax   //add constant 3 to register eax
    - movl %eax, %ebx //move contents of register eax to register ebx
    - movl 8(%ebp), %eax //move contents at memory address (8 + contents(ebp)) //to register eax
    - movl %eax, 4(%ebx,%ecx,4) //effective address is 4 + contents(%ebx) + 4*contents(%ecx)

Key differences from SaM(1) Memory Layout



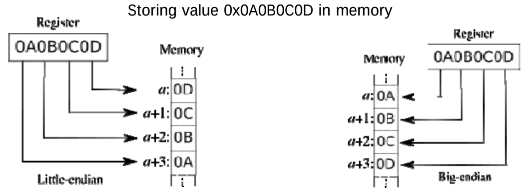
Key differences from SaM (2)

SaM dealt only with 32-bit words and word addresses.

x86 instruction set can address bytes and supports data of different sizes, so you have to be aware of the representation of data.

How are 32-bit quantities stored in memory?

Storing value 0x0A0B0C0D in memory

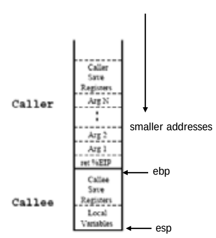


x86 is "little-endian"

Key differences from SaM (3)

- SaM had no general-purpose registers
- x86 has general-purpose registers
  - some registers might be in use when a procedure call is made
  - must save/restore registers so that values in registers survive across the call
- gcc convention:
  - caller save: eax,ecx,edx
  - callee save: ebp,ebx,esi,edi
  - esp is stack pointer
  - important if you want to link your compiled code with C library compiled with gcc

gcc stack frame



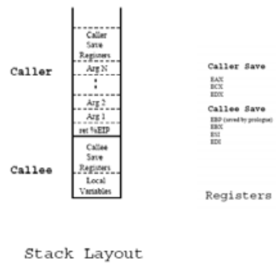
Stack Layout

Similar to SaM stack frame but

- arguments are pushed right to left
  f(arg1,arg2,...,argN)
- registers are saved by caller and callee
- ebp (FBR) is one of callee save registers
- eax is used to return a value from function

Example (gcc)

```
int plus3 (int x) { int res = x + 3; return res; }
int doit (int x) { return plus3 (x); }
int main (void) {  return doit (8); }
_plus3:
        pushl   %ebp              // save ebp
        movl    %esp, %ebp        //ebp points to current frame
        pushl   %esi              //save register esi
        movl    8(%ebp), %esi     //x → esi
        addl    $3, %esi          //esi + 3 → esi
        movl    %esi, %eax        //eax now has return value
        popl    %esi              //restore esi
        movl    %ebp, %esp        //pop local variables
        popl    %ebp              //restore ebp
        ret
_doit:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   8(%ebp)
        call    _plus3
        movl    %ebp, %esp
        popl    %ebp
        ret
_main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $8
        call    _doit
        movl    %ebp, %esp
        popl    %ebp
        ret
```



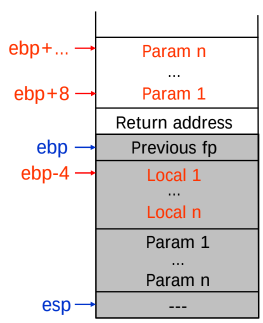
Code Generation

- Three-address code makes it easy to generate assembly
  - Complex expressions in the input program already lowered to sequences of simple IR instructions
  - Just need to translate each low IR instruction into a sequence of assembly instructions

    e.g. a = p+q ⇒ mov 16(%ebp), %ecx
    add 8(%ebp), %ecx
    mov %ecx, -8(%ebp)
- Need to consider many language constructs:
  - Operations: arithmetic, logic, comparisons
  - Accesses to local variables, global variables
  - Array accesses, field accesses
  - Control flow: conditional and unconditional jumps
  - Method calls, dynamic dispatch
  - Dynamic allocation (new)
  - Run-time checks

Accessing Stack Variables

- To access stack variables: use offsets from ebp
- Example:
  8(%ebp) = parameter 1
  12(%ebp) = parameter 2
  -4(%ebp) =local 1



Accessing Stack Variables

- Translate accesses to variables:
  - For parameters, compute offset from %ebp using:
    - Parameter number
    - Sizes of other parameters
  - For local variables, decide on data layout and assign offsets from frame pointer to each local
  - Store offsets in the symbol table
- Example:
  - a: local, offset-4
  - p: parameter, offset+16, q: parameter, offset+8
  - Assignment a = p + q becomes equivalent to:

    -4(%ebp) = 16(%ebp) + 8(%ebp)
  - How to write this in assembly?

Arithmetic

- How to translate: p+q ?
  - Assume p and q are locals or parameters
  - Determine offsets for p and q
  - Perform the arithmetic operation
- Problem: the ADD instruction in x86 cannot take both operands from memory; notation for possible operands:
  - mem32: register or memory 32 bit (similar for r/m8, r/m16)
  - reg32: register 32 bit (similar for reg8, reg16)
  - imm32: immediate 32 bit (similar for imm8, imm16)
  - At most one operand can be mem !
- Translation requires using an extra register
  - Place p into a register (e.g. %ecx): mov 16(%ebp), %ecx
  - Perform addition of q and %ecx: add 8(%ebp), %ecx

Data Movement

- Translate a = p+q:
  - Load memory location (p) into register (%ecx) using a move instr.
  - Perform the addition
  - Store result from register into memory location (a):

```
mov 16(%ebp), %ecx    (load)
add 8(%ebp), %ecx     (arithmetic)
mov %ecx, -8(%ebp)    (store)
```

- Move instructions cannot have two memory operands
  Therefore, copy instructions must be translated using an extra register:
  a = p ⇒ mov 16(%ebp), %ecx
  mov %ecx, -8(%ebp)
- However, loading constants doesn't require extra registers:
  a = 12 ⇒ mov $12, -8(%ebp)

Accessing Global Variables

- Global (static) variables and constants not stack allocated
- Have fixed addresses throughout the execution of the program
  - Compile-time known addresses (relative to the base address where program is loaded)
  - Hence, can directly refer to these addresses using symbolic names in the generated assembly code
- Example: string constants

  str: .string "Hello world!"

  - The string will be allocated in the static area of the program
  - Here, "str" is a label representing the address of the string
  - Can use $str as a constant in other instructions:

  push $str

Accessing Heap Data

- Heap data allocated with new (Java) or malloc (C/C++)
  - Such allocation routines return address of allocated data
  - References to data stored into local variables
  - Access heap data through these references
- Array accesses in language with dynamic array size
  - access a[i] requires:
    - Compute address of element: a + i * size
    - Access memory at that address
  - Can use indexed memory accesses to compute addresses
  - Example: assume size of array elements is 4 bytes, and local variables a, i (offsets –4, -8)

```
a[i] = 1  ⇒  mov –4(%ebp), %ebx       (load a)
             mov –8(%ebp), %ecx       (load i)
             mov $1, (%ebx,%ecx,4)    (store into the heap)
```

Control-Flow

- Label instructions
  - Simply translated as labels in the assembly code
  - E.g., label2: mov $2, %ebx
- Unconditional jumps:
  - Use jump instruction, with a label argument
  - E.g., jmp label2
- Conditional jumps:
  - Translate conditional jumps using test/cmp instructions:
  - E.g., tjump b L

```
cmp %ecx, $0
jnz L
```

    where %ecx hold the value of b, and we assume booleans are represented as 0=false, 1=true

Run-time Checks

- Run-time checks:
  - Check if array/object references are non-null
  - Check if array index is within bounds
- Example: array bounds checks:
  - if v holds the address of an array, insert array bounds checking code for v before each load (...=v[i]) or store (v[i] = ...)
  - Assume array length is stored just before array elements:

```
cmp $0, -12(%ebp)          (compare i to 0)
jl ArrayBoundsError        (test lower bound)
mov –8(%ebp), %ecx         (load v into %ecx)
mov –4(%ecx), %ecx         (load array length into %ecx)
cmp –12(%ebp), %ecx        (compare i to array length)
jle ArrayBoundsError       (test upper bound)
. . .
```


X86 Assembly Syntax

- Two different notations for assembly syntax:
  - AT&T syntax and Intel syntax
  - In the examples: AT&T (gcc) syntax
- Summary of differences:

	AT&T	Intel
Order of operands	op a, b : b is destination	op a, b : a is destination
Memory addressing	disp(base,offset,scale)	[base + offset*scale + disp]
Size of memory operands	instruction suffixes (b,w,l) (e.g., movb, movw, movl)	operand prefixes (byte ptr, word ptr, dword ptr)
Registers	%eax, %ebx, etc.	eax, ebx, etc.
Constants	$4, $foo, etc	4, foo, etc