Compiler construction

Lecture 6: Code generation for x86

Thomas Sewell

Spring 2020

Chalmers University of Technology — Gothenburg University

This lecture

- x86 architecture
- Calling conventions
- Some x86 instructions
- Compiler backends
    - From LLVM to assembler
    - Instruction selection
    - Instruction scheduling
    - Register allocation

x86 architecture

x86: assembly for a real machine

High-level view of x86 assembly

- Arithmetic operations done with values in registers
- Long term storage in memory
- Little support for function calls; programmer must handle return addresses, jumps, stack frames, etc.
- Code is assembled and run; no further optimization
    - Bad assembly means bad performance
- x86 is a CISC architecture (Complex Instruction Set)
    - Usually few registers
    - Complicated instruction arguments

x86 assembler, a first example

JavaLette (or C)

```
> cat ex1.jl

 int f (int x, int y) {
   int z = x + y;
   return z;
 }
```

This might be compiled to the assembler code to the right.

NASM assembly code

```
segment .text
        global f
f:
        push dword ebp
        mov  ebp, esp
        sub  esp, 4
        mov  eax, [ebp+12]
        add  eax, [ebp+8]
        mov  [ebp-4], eax
        mov  eax, [ebp-4]
        mov  esp, ebp
        pop  ebp
        ret
```

Example explained

NASM code commented

```
segment .text                  ; code area
        global f               ; f has external scope
f:                             ; entry point for f
        push dword ebp         ; save caller's fp
        mov  ebp, esp          ; set our fp
        sub  esp, 4            ; allocate space for z
        mov  eax, [ebp+12]     ; move y to eax
        add  eax, [ebp+8]      ; add x to eax
        mov  [ebp-4], eax      ; move eax to z
        mov  eax, [ebp-4]      ; return value to eax
        mov  esp, ebp          ; restore caller's sp
        pop  ebp               ; restore caller's fp
        ret                    ; pop return addr, jump
```

Intel x86 architectures

Long history

8086	1978. First IBM PCs, 16 bit registers, real mode
80286	1982. AT, Windows, protected mode
80386	1985. 32 bit registers, virtual memory
80486	(and Pentium, Pentium II-IV) 1989 – 2003. Math coprocessor, pipelining, caches, SSE, ...
Intel Core 2	2006. Multi-core
Core i3/i5/i7	2009 —

Backwards compatibility: sold as “IBM PC-compatible”.
Lots of legacy features.

Produced by Intel, and compatible chips from AMD and others.

Which system should you target?

For the “Part C: native code” extension, you may pick a target. It makes sense to pick hardware you have, e.g. an x86 (your computer) or ARM (your phone or tablet).

A Compiler Target

- An ISA (Instruction Set Architecture) - what instructions exist
- Calling Convention - rules for how to use stack/registers

The hardware, libraries, OS, program loader and libc (base library) all together define the environment the compiled program runs in.

Which version should you target?

x86

When speaking of the x86 architecture, one generally means register/instruction set for the 80386 (with floating-point operations).

You can compile code which would run on a 386. It can also run on a modern x86 machine.

You can also use a more modern 32-bit x86 code, using SSE instructions.

The following slides will present x86-based instructions. You can run this code on a 64-bit machine by providing some extra library code. However, many students may prefer to target an x86-64 machine in 64-bit mode, which is similar but different (more on that later).

x86 registers

General purpose registers (32-bits)

EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI.

Conventional use:

EBP and ESP for frame pointer and stack pointer.

Segment registers

Legacy from old segmented addressing architecture.

Can be ignored in JAVALETTE compilers.

Floating-point registers

Eight 80–bit registers ST0 – ST7 organised as a stack.

Flag registers

Status registers with bits for results of comparisons, etc.

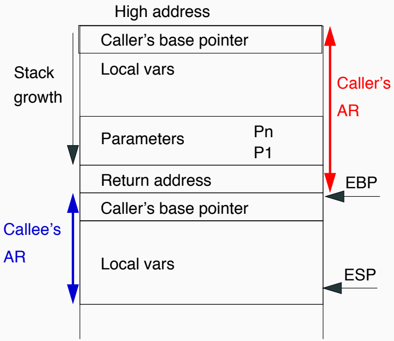
We will discuss these later.

Calling convention

Data area for parameters and local variables

Runtime stack

- Contiguous memory area
- Grows from high addresses downwards
- AR layout illustrated
- EBP contains current base pointer (= frame pointer)
- ESP contains current stack pointer
- Note: We need to store return address (address of instruction to jump to on return)



Calling convention

Caller, before call

- Push params (in reverse order)
- Push return address
- Jump to callee entry

```
push dword param_n
...
push dword param_1
call f
```

Calling convention

Caller, before call

- Push params (in reverse order)
- Push return address
- Jump to callee entry

```
push dword param_n
...
push dword param_1
call f
```

Callee, on entry

- Push caller's base pointer
- Update current base pointer
- Allocate space for locals

```
push dword ebp
mov  ebp, esp
sub  esp, localbytes
```

Calling convention

Caller, before call

- Push params (in reverse order)
- Push return address
- Jump to callee entry

```
push dword param_n
...
push dword param_1
call f
```

Callee, on entry

- Push caller's base pointer
- Update current base pointer
- Allocate space for locals

```
push dword ebp
mov  ebp, esp
sub  esp, localbytes
```

Callee, on exit

- Restore base and stack ptr
- Pop return address and jump

```
mov  esp, ebp
pop  ebp
ret
```

Calling convention

Caller, before call

- Push params (in reverse order)
- Push return address
- Jump to callee entry

```
push dword param_n
...
push dword param_1
call f
```

Caller, after call

- Pop parameters

```
add esp parambytes
```

Callee, on entry

- Push caller's base pointer
- Update current base pointer
- Allocate space for locals

```
push dword ebp
mov  ebp, esp
sub  esp, localbytes
```

Callee, on exit

- Restore base and stack ptr
- Pop return address and jump

```
mov  esp, ebp
pop  ebp
ret
```

Calling convention

Caller, before call

- Push params (in reverse order)
- Push return address
- Jump to callee entry

```
push dword param_n
...
push dword param_1
call f
```

Caller, after call

- Pop parameters

```
add esp parambytes
```

Callee, on entry

- Push caller’s base pointer
- Update current base pointer
- Allocate space for locals

```
enter localbytes, 0
```

Callee, on exit

- Restore base and stack ptr
- Pop return address and jump

```
leave
ret
```

Parameters, local variables and return values

Parameters

- In the callee code, integer parameter 1 has address EBP+8, parameter 2 EBP+12, etc.
- Parameter values accessed with indirect addressing: [EBP+8], etc.
- Double parameters require 8 bytes
- Here EBP+n means "(address stored in EBP) + n"

Local variables

- First local var is at address EBP−4, etc.
- Local vars are conventionally addressed relative to EBP, not ESP
- Again, refer to vars by indirect addressing: [EBP−4], etc.

Return values

Integer and boolean values are returned in EAX, doubles in ST0

Register usage

Scratch registers (caller save)

EAX, ECX and EDX must be saved by caller before call, if used; can be freely used by callee.

Callee save register

EBX, ESI, EDI, EBP, ESP.

For EBP and ESP, this is handled in the code patterns.

Note

- What we have described is one common calling convention for 32-bit x86, called cdecl
- Other conventions exist, e.g. 64-bit System V and Microsoft
- `https://en.wikipedia.org/wiki/X86_calling_conventions`

Assemblers for x86

Several alternatives

- Several assemblers for x86 exist, with different syntax
- These slides use NASM, the Netwide Assembler, which is available for several platforms
  - You should use an assembler, but it doesn't have to be NASM.
- We also recommend Paul Carter's book and examples; follow link from course website
- Some syntax differences to the GNU assembler:
  - GNU uses `%eax` etc. as register names
  - For two-argument instructions, the operands have opposite order!
  - Different syntax for indirect addressing

If you use `gcc -S ex.c`, you will get GNU syntax

Example: GNU syntax

First example, revisited

```
> gcc -c ex1.c
> objdump -d ex1.o

ex1.o:     file format elf32-i386
Disassembly of section .text:

00000000 <f>:
   0:        55              push   %ebp
   1:        89 e5           mov    %esp,%ebp
   3:        8b 45 0c        mov    0xc(%ebp),%eax
   6:        03 45 08        add    0x8(%ebp),%eax
   9:        c9              leave
   a:        c3              ret
```

Assembler

Integer arithmetic; two-adress code

Addition, subtraction and multiplication

```
add  dest, src  ; dest := dest + src
sub  dest, src  ; dest := dest - src
imul dest, src  ; dest := dest * src
```

Operands can be values in registers or in memory; `src` also a literal.

Division – one-address code

```
idiv denom
(eax, edx) := ((edx:eax) / denom, (edx:eax) % denom)
```

- The numerator is the 64-bit value EDX:EAX (no other choices)
- Both div and mod are performed; results in EAX resp. EDX
- EDX must be zeroed before division

Example

JavaLette program

```
int main () {
  printString "Input a number: ";
  int n = readInt();
  printInt(2 * n);
  return 0;
}
```

The above code could be translated as follows (slightly optimized to fit on slide).

Code for `main`

```
push dword ebp
mov ebp, esp
push str1
call printString
add esp, 4
call readInt
imul eax, 2
push eax
call printInt
add esp, 4
mov eax, 0
leave
ret
```

Example, continued

Complete file

```
extern printString, printInt
extern readInt

segment .data
str1 db "Input a number: "

segment .text
  global main

main:
; code from previous slide
```

Comments

- IO functions are external; we will come back to that
- The `.data` segment contains constants such as `str1`
- The `.text` segment contains code
- The `global` declaration gives `main` external scope (can be called from code outside this file)

Floating-point arithmetic in x86

FP Option 1: `ST0`-`ST7` 80-bit stack

Stacking numbers

`fld src`	Pushes value in `src` on fp stack
`fild src`	Pushes integer value in `src` on fp stack
`fstp dest`	Stores top of fp stack in `dest` and pops

Both `src` and `dest` can be fp register or memory reference.

Arithmetic (selection)

`fadd src`	Adds `src` to `ST0` (also `fsub`, `fmul`, `fdiv`)
`fadd to dest`	Adds `ST0` to `dest`
`faddp dest`	Adds `ST0` to `dest`, then pop

Floating-point arithmetic in SSE2

FP Option 2:

New registers

- 128-bit registers `XMM0`–`XMM7` (later also `XMM8`–`XMM15`)
- Each can hold two double precision floats or four single-precision floats
- SIMD operations for arithmetic
  - Single Instruction, Multiple Data

Arithmetic instructions

- Two-address code, ADDSD, MULSD, etc.
- SSE2 fp code similar to integer arithmetic

History

The original 8086 processor was extended with a companion 8087 floating point processor.

- Same generation but more novel/unusual design
- Values must move from integer to FP registers via memory, which simplifies the interface between the processors
- Novel ideas (stack model, 80-bit arithmetic) dropped in later history

Control flow

Integer comparisons

- `cmp` v_1 v_2
- $v_1 - v_2$ is computed and bits in the flag register are set:
  - `ZF` is set iff value is zero
  - `OF` is set iff result overflows
  - `SF` is set iff result is negative

Control flow

Integer comparisons

- `cmp` v_1 v_2
- $v_1 - v_2$ is computed and bits in the flag register are set:
    - `ZF` is set iff value is zero
    - `OF` is set iff result overflows
    - `SF` is set iff result is negative

Branch instructions (selection)

- `JZ lab` branches if `ZF` is set
- `JL lab` branches if `SF` is set
- Similarly for the other relations between v_1 and v_2
- `fcomi src` compares `ST0` and `src` and sets flags; can be followed by branching as above

One more example

JAVALETTE (or C)

```
int sum(int n) {
  int res = 0;
  int i   = 0;
  while (i < n) {
    res = res + i;
    i++;
  }
  return res;
}
```

Naive assembler

```
sum:    enter 8, 0
        mov [ebp-4], 0
        mov [ebp-8], 0
        jmp L2
L3:     mov eax, [ebp-8]
        add [ebp-4], eax
        inc [ebp-8]
L2:     mov eax, [ebp-8]
        cmp eax, [ebp+8]
        jl L3
        mov eax, [ebp-4]
        leave
        ret
```

How to do an x86 backend

Starting point

Two alternatives:

- From LLVM code (requires your basic backend to generate LLVM code as a data structure, not directly as strings); will generate many local vars
- From AST's generated by the frontend (means a lot of code common with LLVM backend)

Variables

In either case, your code will contain a lot of variables/virtual registers. Possible approaches:

- Keep values on the stack, storing and fetching at each access; gives really slow code (`x = y + z;` 3x memory, 1x arithmetic)
- Do register allocation[1]; much better code

[1]Future lecture

Input and output

Linux building

To assemble a NASM file to `file.o`:

```
nasm -f elf file.asm
```

To link:

```
gcc file.o <runtime>
```

(or `clang file.o <runtime>`)

Result is executable `a.out`

Runtime Options

- Define `printInt` etc in C, `<runtime> = runtime.c`
- Compile to `runtime.ll` then `runtime.s`, compatible with `gcc`
- x86-64 `runtime.s` provided in `tda283` repo

More info

Paul Carter's book (link on course web site) gives more info.

From LLVM to assembler

From LLVM to assembler

Several stages

- Instruction selection
- Instruction scheduling
- SSA-based optimizations (lots of these!)
- Register allocation
- Prolog/epilog code (AR management)
- Code emission

Target-independent generation

Wherever possible a multi-target compiler implements general versions of these stages with target-specific settings.

From LLVM to assembler

Several stages

- Instruction selection
- Instruction scheduling
- SSA-based optimizations (lots of these!)
- Register allocation
- Prolog/epilog code (AR management)
- Code emission

Target-independent generation

Wherever possible a multi-target compiler implements general versions of these stages with target-specific settings.

A Note

These slides (& register allocation next week) will describe industrial approaches. Your approach can be simpler†.

Native code generation, revisited

More complications

We have been ignoring some important concerns:

- The instruction set in real-world processors typically offer many different ways to achieve the same effect. Thus, when translating an IR program to native code we must do instruction selection, i.e., choose between available alternatives.
- Often an instruction sequence contain independent parts that can be executed in arbitrary order. Different orders may take very different time; thus a code generator should do instruction scheduling.

Both these task are complex and interact with register allocation.

In LLVM, these tasks are done by the native code generator `llc` and the JIT compiler in `lli`.

Instruction selection

Further observations

- Instruction selection for RISC machines generally simpler than for CISC machines
    - However most RISC architectures have some complex instructions
- The number of translation possibilities grow (combinatorially) as one considers larger chunks of IR code for translation

Pattern matching

The IR code can be seen as a pattern matching problem: The native instructions are seen as patterns; instruction selection is the problem to cover the IR code by patterns.

Instruction selection

Further observations

- Instruction selection for RISC machines generally simpler than for CISC machines
    - However most RISC architectures have some complex instructions
- The number of translation possibilities grow (combinatorially) as one considers larger chunks of IR code for translation

Pattern matching

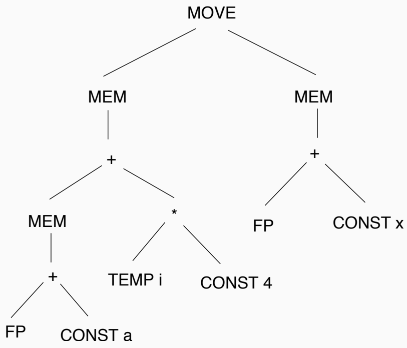
The IR code can be seen as a pattern matching problem: The native instructions are seen as patterns; instruction selection is the problem to cover the IR code by patterns.

Two approaches

- Tree pattern matching: think of IR code as tree
- Peephole matching: think of IR code as sequence

Tree pattern matching, an example

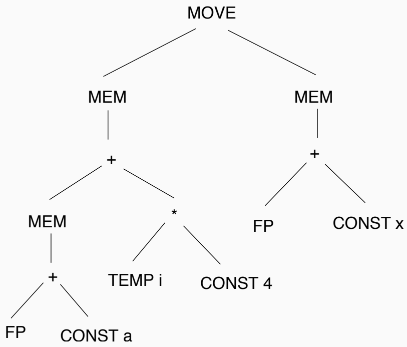
`a[i] := x` **as tree IR code**



- `a` and `x` local vars, `i` in register
- `a` is pointer to first element

Tree pattern matching, an example

`a[i] := x` as tree IR code



- `a` and `x` local vars, `i` in register
- `a` is pointer to first element

Algorithm outline

- Represent native instructions as patterns, or tree fragments
- Tile the IR tree using these patterns so that all nodes in the tree are covered
- Output the sequence of instructions corresponding to the tiling

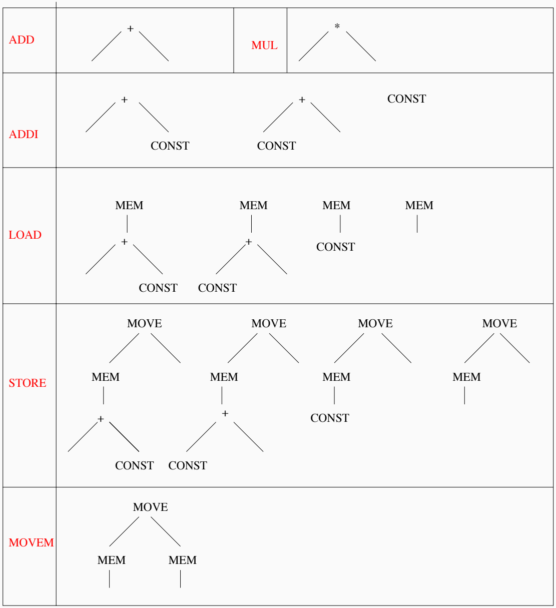
A simple instruction set

ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j - c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

Notes

- We consider only arithmetic and memory instructions (no jumps!)
- Basic x86 is more complicated
- SIMD instructions (also on RISC machines) even more complex

Identifying patterns (incomplete)



Instruction scheduling, background

Simple-minded, old-fashioned view of processor

Fetch an instruction, decode it, fetch operands, perform operation, store result. Then fetch next operation, ...

Modern processors

- Several instructions under execution concurrently
- Either registers or operations can be waiting for data
- Memory system is main cause of delays
- Some arithmetic operations may also take several cycles

Consequence

Important to understand data dependencies and order instructions advantageously. Also, test your optimisations.

Instruction scheduling, example

Example (from Cooper)

```
w = w * 2 * x * y * z
```

- Memory op takes 3 cycles, mult 2 cycles, add one cycle
- One instruction can be issued each cycle, if data available

Instruction scheduling, example

Example (from Cooper)

```
w = w * 2 * x * y * z
```

- Memory op takes 3 cycles, mult 2 cycles, add one cycle
- One instruction can be issued each cycle, if data available

Schedule 1

```
r1 <- M [fp + @w]
r1 <- r1 + r1
r2 <- M [fp + @x]
r1 <- r1 * r2
r2 <- M [fp + @y]
r1 <- r1 * r2
r2 <- M [fp + @z]
r1 <- r1 * r2
M [fp + @w] <- r1
```

Instruction scheduling, example

Example (from Cooper)

```
w = w * 2 * x * y * z
```

- Memory op takes 3 cycles, mult 2 cycles, add one cycle
- One instruction can be issued each cycle, if data available

Schedule 1

```
r1 <- M [fp + @w]
r1 <- r1 + r1
r2 <- M [fp + @x]
r1 <- r1 * r2
r2 <- M [fp + @y]
r1 <- r1 * r2
r2 <- M [fp + @z]
r1 <- r1 * r2
M [fp + @w] <- r1
```

Schedule 2

```
r1 <- M [fp + @w]
r2 <- M [fp + @x]
r3 <- M [fp + @y]
r1 <- r1 + r1
r1 <- r1 * r2
r2 <- M [fp + @z]
r1 <- r1 * r3
r1 <- r1 * r2
M [fp + @w] <- r1
```

Instruction scheduling

Comments

- Problem is NP-complete for realistic architectures
- Common technique is list scheduling: greedy algorithm for scheduling a basic block
- Builds graph describing data dependencies between instructions and schedules instructions from ready list of instructions with available operands

Interaction

Despite interaction between selection, scheduling and register allocation, these are typically handled independently (and in this order).

x86-64

x86 has a long history of extension

- ESP $\neq$ extra sensory perception
- ESP $\equiv$ Extended (32-bit) stack pointer
- bottom 16 bits of ESP are the old (8086) 16-bit SP

x86-64 continues

- R8-R15 are new in AMD64/x86-64, R stands for ‘register’
- RSP, RBP are 64-bit registers
- bottom 32 bits of RSP are ESP, bottom 16 bits are SP, etc
- look up a table of registers, synonyms etc
- substitute ebp/rbp esp/rsp from previous slides

Demo?

x86/native backend extension

Comments

- Two credits
- Three requirements
    - Working native code
    - Need to implement at least one optimization pass
    - Need to try to use registers
- Acts as a ‘multiplier’ for other extensions