Compiler construction 2011

Lecture 4

- x86 architecture
- Calling conventions
- Some x86 instructions

x86: assembly for a real machine

First comparison with JVM

- Not a stack machine; no correspondence to operand stacks.
- Arithmetic etc are instead done with values in **registers**.
- Much more limited support for function calls; you need to handle return addresses, jumps, allocation of stack frames etc yourself.
- Your code is assembled and run; no further optimization.
- CISC architecture with few registers. Straightforward code will run slowly.

x86 assembler, a first example

Javalette (or C)

```
> more ex1.c
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]
        leave
        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
        leave              ; restore caller's fp/sp
        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, Pentium, Pentium II, III, IV. 1989 – 2003.
  Math coprocessor, pipelining, caches, SSE . . .
- Intel Core 2. 2006. Multi-core.
- Core i3/i5/i7. 2009/10.

Backwards compatibility important; leading to a large set of opcodes.

Not only Intel offer x86 processors: also AMD is in the market.

Which version should you target?

x86

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

You can compile code which would run on a 386
– or you may use SSE2 operations for a more recent version.

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

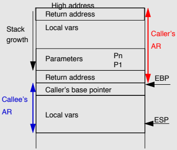
Status registers with bits for results of comparisons, etc.
We discuss these later.

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).

Illustration



Calling convention

Caller, before call

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

Code pattern:

```
push dword paramn
...
push dword param1
call f
```

Caller, after call

Pop parameters.

Code pattern:

```
add esp parambytes
```

Callee, on entry

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

Code pattern:

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

Callee, on exit

Restore base and stack pointer.
Pop return address and jump.

Code pattern:

```
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.
Parameters are conventionally addressed relative to `ebp`, not `esp`.

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, but we omit them.

Assemblers for x86

Several alternatives

- Several assemblers for x86 exist, with different syntax.
- We will use NASM, the Netwide Assembler, which is available for several platforms.
- We also recommend Paul Carter's book and examples. Follow link from course web site.
- 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
>
```


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.
  Trick: `xor edx, edx`.

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 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

Moving numbers (selection)

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

Arithmetic (selection)

`fadd` *src* — *src* added to `ST0`.
`fadd to` *dest* — `ST0` added to *dest*.
`faddp` *dest* — `ST0` added to *dest*, then pop.
Similar variants for `fsub`, `fmul` and `fdiv`.

Floating-point arithmetic in SSE2

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.

Arithmetic instructions

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

Control flow

Integer comparisons

`cmp` *v1 v2*
v1-v2 is computed and bits in the flag registers 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 *v1* and *v2*.

`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:    push dword ebp
        mov ebp, esp
        sub esp, 8
        mov [ebp-4], 0
        mov [ebp-8], 0
        jmp L2
L3:     mov eax, [ebp-4]
        add [ebp-8], eax
        inc [ebp-4]
L2:     mov eax, [ebp-4]
        cmp eax, [ebp+8]
        jl L3
        mov eax, [ebp-8]
        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 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:

- Treat these as local vars, storing to and fetching from stack at each access. Gives really slow code.
- Do (linear scan) register allocation. Much better; you will want to do this if you choose do do this backend.

Input and output

A simple proposal

Define `printInt`, `readInt` etc in C. Then link this file together with your object files using `gcc`.

Alternative: Compile `runtime.ll` with `llvm-as` and `llc` to get `runtime.s`; this can be given to `gcc` as below.

Linux building

To assemble a NASM file to `file.o`:
`nasm -f elf file.asm`
To link:
`gcc file.o runtime.c`
Result is executable `a.out`.

More info

Paul Carter's book (link on course web site) gives more info.
His driver and input routines could possibly be used, but the above seems better.