Compiler construction 2012

Lecture 7

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

x86: assembly for a real machine

First comparison with JVM

- Not a stack machine; no direct correspondence to operand stacks.
- Arithmetics etc is 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

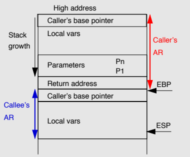
Status registers with bits for results of comparisons, etc.
We discuss these later.

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.
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, 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* $\cdot$ *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-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 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.

From LLVM to assembler

Several stages

- Instruction selection.
- Instruction scheduling.
- SSA-based optimizations.
- Register allocation.
- Prolog/epilog code (AR management).
- Late optimizations.
- Code emission.

Target-independent generation

Also much of this is done in target-independent ways and using general algorithms operating on target descriptions.

Native code generation, revisited

More complications

So far, we have ignored some important concerns in code generation:

- 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 the code generator must 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.
- The number of translation possibilities grow (combinatorially) as one considers larger chunks of IR code for translation.

Pattern matching

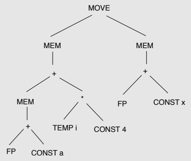
The IR code can be seen as a pattern matching problem: The native instructions are seen as patterns; instruction selection is the problem to cover the IR code by patterns.

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 (from Appel)



a and x local vars, i in register.
a is pointer to first array 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.

Two variants

- Greedy algorithm (top down).
- Dynamic programming (bottom up); based on cost estimates.

A simple instruction set

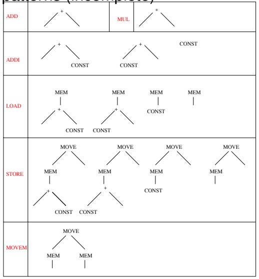
ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j - c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

Notes

- We consider only arithmetic and memory instructions (no jumps!).
- Assume special register r_0, which is always 0.

Example done in class.

Identifying patterns (incomplete)



Peephole matching

Recall: peephole optimization

Code improvement by local simplification of the code within a small sliding window of instructions.

Can be used for instruction selection

Often one further intermediate language between IR and native code; peephole simplification done for that language.

Retargetable compilers

Instruction selection part of compiler generated from description of target instruction set (**code generator generators**).

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.
- Memory system cause delays, with operations waiting for data.
- Similar problems for results from arithmetic operations, that may take several cycles.

Consequence

Important to understand data dependencies and order instructions advantageously.

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).

Register allocation

An important code transformation

When translating an IR with (infinitely many) virtual registers to code for a real machine, we must

- assign virtual registers to physical registers.
- write register values to memory (**spill**), at program points when the number of live virtual registers exceeds the number of available registers.

Register allocation is very important; good allocation can make a program run an order of magnitude faster (or more) as compared to poor allocation.

The interference graph

Live sets and register usage

A variable is **live** at a point in the CFG, if it may be used in the remaining code without previous assignment.

If two variables are live at the same point in the CFG, they must be in different registers.

Conversely, two variables that are never live at the same time can share a register.

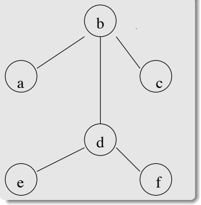
Interfering variables

We say that variables `x` and `y` **interfere** if they are both live at some point.

The **interference graph** has variables as nodes and edges between interfering variables.

An example

How many registers are needed?



Answer: Two!

Use one register for a, c and d, the other for b, e and f.

Reformulation

To assign K registers to variables given an interference graph can be seen as colouring the nodes of the graph with K colours, with adjacent nodes getting different colours.

Register allocation by graph colouring

The algorithm (K colours available)

1. Find a node n with less than K edges. Remove n and its edges from the graph and put on a stack.
2. Repeat with remaining graph until either
   - only K nodes remain **or**
   - all remaining nodes have at least K adjacent edges.

   In the first case, give each remaining node a distinct colour and pop nodes from the stack, inserting them back into the graph with their edges and colouring them.

   In the second case, we may need to **spill** a variable to memory. Optimistic algorithm: Choose one variable and push on the stack. Later, when popping the stack, we **may** be lucky and find that the neighbours use at most K-1 colours.

Complexity

A hard problem

The problem to decide whether a graph can be K-coloured is NP-complete.

The simplify/select algorithm on the previous slide works well in practice; its complexity is $O(n^2)$, where n is the number of virtual registers used.

When optimistic algorithm fails, memory store and fetch instructions must be added and algorithm restarted.

Heuristics to choose variable to spill:

- Little use+def within loop;
- Interference with many variables.

Move instructions

An example

```
t := s
x := s + 1
y := t + 2
...
```

`s` and `t` interfere,
but if `t` is not later redefined, they may share a register.

Coalescing

Move instructions `t := s` can sometimes be removed and the nodes `s` and `t` merged in the interference graph.

Conditions:

- No interference between `s` and `t` for other reasons.
- The graph must remain colourable. Safe strategies exist.

Linear scan register allocation

Compilation time vs code quality

Register allocation based on graph colouring produces good code, but requires significant compilation time.
For e.g. JIT compiling, allocation time is a problem.
The Java HotSpot compiler uses a **linear scan** register allocator.

Much faster and in many cases only 10% slower code.

The linear scan algorithm

Preliminaries

- Number all the instructions 1, 2, . . . in some way
  (for now, think of numbering them from top to bottom).
  (Other instruction orderings improves the algorithm; also here depth first ordering is recommended.)
- Do a simplified liveness analysis, assigning a **live range** to each variable.
  A live range is an interval of integers starting with the number of the instruction where the variable is first defined and ending with the number where it is last used.
- Sort live ranges in order of increasing start points into list L.

The linear scan algorithm

The algorithm

- Maintain a list, called *active*, of live ranges that have been assigned registers. *active* is sorted by increasing end points and initially empty.

  Traverse L and for each interval I:
  - Traverse *active* and remove intervals with end points before start point of I.
  - If length of *active* is smaller than number of registers, add I to *active*; otherwise spill either I or the last element of *active*.

    In the latter case, the choice of interval to spill is usually to keep interval with longest remaining range in *active*.

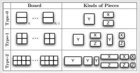
More algorithms

Still a hot topic

Register allocation is still an active research area, an indication of its importance in practice.

Puzzle solving

Recent work by Pereira and Palsberg views register allocation as a puzzle solving problem.



Chordal graphs

Hack, Grund and Goos exploit the fact that the interference graph is **chordal** to get an $O(n^2)$ optimal algorithm.
Care is needed when destructing SSA form.

Summing up

What's needed?

To claim credits for a x86 backend, straightforward code generation is enough. Consider register allocation.

But we require correct code and runnable programs; “almost done” does not give credit.

Next time

Guest lecture by Josef Svenningsson on compiling techniques for Feldspar, a Haskell-based DSL for signal processing, targetting highly optimized C code for DSP processors.