Introduction to Code Generation

- We first focus on generating code for a stack machine with accumulator
- We want to run the resulting code on a real machine, e.g., the MIPS/x86 processor (or simulator)
    - The ideas do not change much for abstract machines like LLVM IR abstract machine, except some issues like register-allocation.
- We simulate stack machine instructions using MIPS/x86 instructions and registers.

MIPS architecture

- Prototypical Reduced Instruction Set Computer (RISC)
- Most operations use registers for operands and results
- Use `load` and `store` instructions to use values in memory
- 32 general purpose registers (32 bits each)
    - We will use only three registers: `$sp`, `$a0`, and `$t1` (a temporary register)

x86 architecture

- Complex Instruction Set Computer (CISC)
- Significantly larger opcode set : 400-odd compared to 40-odd in RISC
- Opcodes often can operate on both registers and/or memory
    - Do not necessarily need separate load/store instructions
- 8 general purpose registers (32 bits each)
    - We will use `%esp`, `%eax`, and `%ecx` (a temporary register)
    - Intel engineers felt that it is better to provide more opcodes and less registers. Use on-chip real-estate for more functional units and logic (by saving space through a shorter register file and its connections).

Need some code-generation invariants

- The accumulator is kept in MIPS register `$a0` (or x86 register `%eax`)
- The stack is kept in memory
    - Grows downwards towards lower addresses
    - Standard convention on both MIPS and x86
- For MIPS
    - The next location of the stack is kept in MIPS register `$sp`. The top of the stack is at address `$sp+4`
- For x86
    - The top of the stack is at address `%esp`. The next location of the stack is at address `%esp-4`.
- The name "`sp`" stands for stack-pointer.

MIPS opcodes (relevant only)

- `lw reg1, offset(reg2)`
    - Load 32-bit word from address `reg2+offset` into `reg1`
- `add reg1, reg2, reg3`
    - `reg1 <-- reg2+reg3`
- `sw reg1, offset(reg2)`
    - Store 32-bit word `reg1` to address `reg2+offset`
- `addiu reg1, reg2, imm`
    - `reg1 <-- reg2+imm`
    - "`u`" means overflow is not checked (overflow means different things for signed and unsigned interpretation of the registers)
- `li reg, imm`
    - `reg <-- imm`

x86 opcodes (relevant only)

- `movl %reg1/(memaddr1)/$imm, %reg2/(memaddr2)`
    - Move 32-bit word from register `reg1` (or address `memaddr1` or the immediate value itself) into `reg2` or to memory address `memaddr2`

    - Captures several opcodes in one mnemonic (load, store, li, move-register, etc.). More powerful than RISC, e.g., MIPS cannot move immediate value directly to memory
- `add %reg1/(memaddr1)/$imm, %reg2/(memaddr2)`
    - `%reg2/(memaddr2) <-- reg1/(memaddr1)/imm + %reg2/(memaddr2)`
    - Overflow is always computed for both signed/unsigned arithmetic. Happens in parallel so not in critical performance path, but switches more transistors (more power)
- `push %reg/(memaddr)/$imm`
    - `(%esp-4) <-- reg/(memaddr)/imm; %esp <-- %esp-4`
- `pop %reg/(memaddr)/$imm`
    - `reg/(memaddr)/imm <-- (%esp); %esp <-- %esp+4`
- push/pop are "higher-level" opcodes: enables faster execution paths for these common operations

The stack-machine code for `7+5` in MIPS:

```
acc <-- 7                       : li $a0, 7
push acc                           : sw $a0, 0($sp)
                                     addiu $sp, $sp, -4
acc <-- 5                       : li $a0, 5
acc < acc + top_of_stack        : lw $t1, 4($sp)
                                   : add $a0, $a0, $t1
pop                                : addiu $sp, $sp, 4
```

The stack-machine code for `7+5` in x86:

```
acc <-- 7                       : movl $7,%eax
push acc                           : pushl %eax
acc <-- 5                       : movl $5, %eax
acc < acc + top_of_stack        : addl (%esp),%eax
pop                                : popl %ecx    #just pop the register to unused register ecx
```

A more optimized version was possible in x86:

```
acc <-- 7                       : push $7
push acc
acc <-- 5                       : mov $5, %eax
acc < acc + top_of_stack        : add (%esp), %eax
pop                                : pop %ecx    #just pop the register to unused register ecx
```