中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第21讲：目标代码生成(1)

张献伟

xianweiz.github.io

DCS290, 6/2/2022

Review Questions

- Q1: what is a Basic Block?

  A straight-line sequence of code with only one entry point and only one exit.

- Q2: how to partition code into BBs?

  Identify leader insts; a BB consists of a leader inst and subsequent insts before next leader.

- Q3: BBs of the listed code?

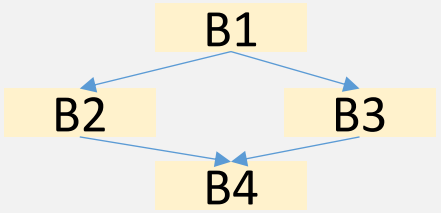
  B1, B2, B3, B4

- Q4: What is a control-flow graph?

  A directed graph where nodes are BBs, edges show flow of execution between BBs.

- Q5: What is the CFG of the listed code?

```
     w = 0          } B1
     y = 0
     x = x + y
     if x > z: goto L1
     y = z          } B2
     z ++
     goto L2
L1:  y = x          } B3
     x ++
L2:  w = x + z      } B4
```

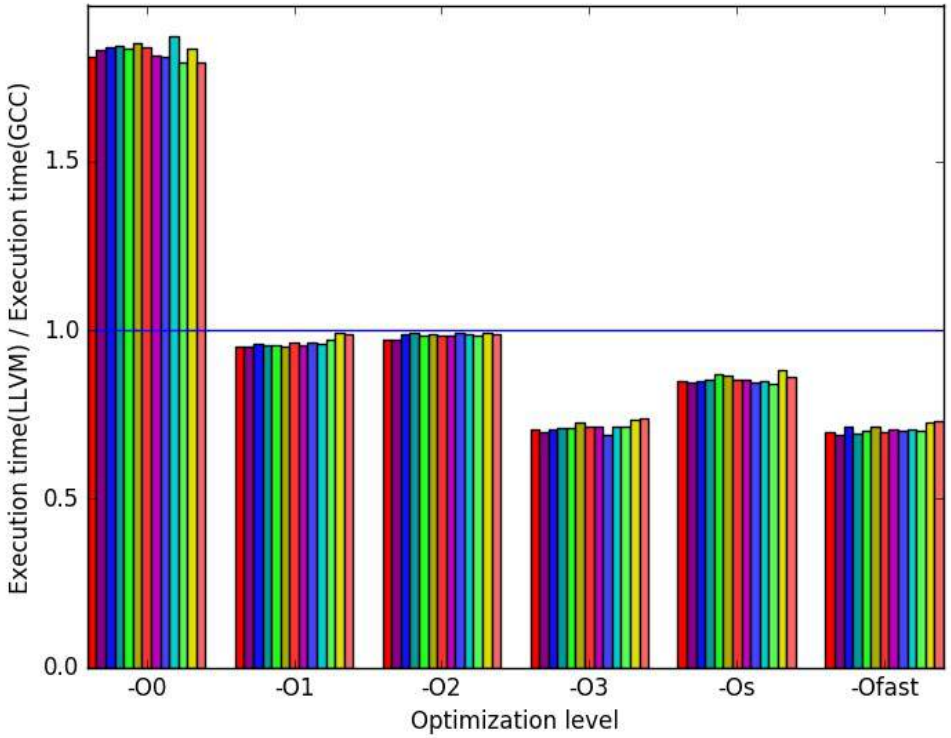


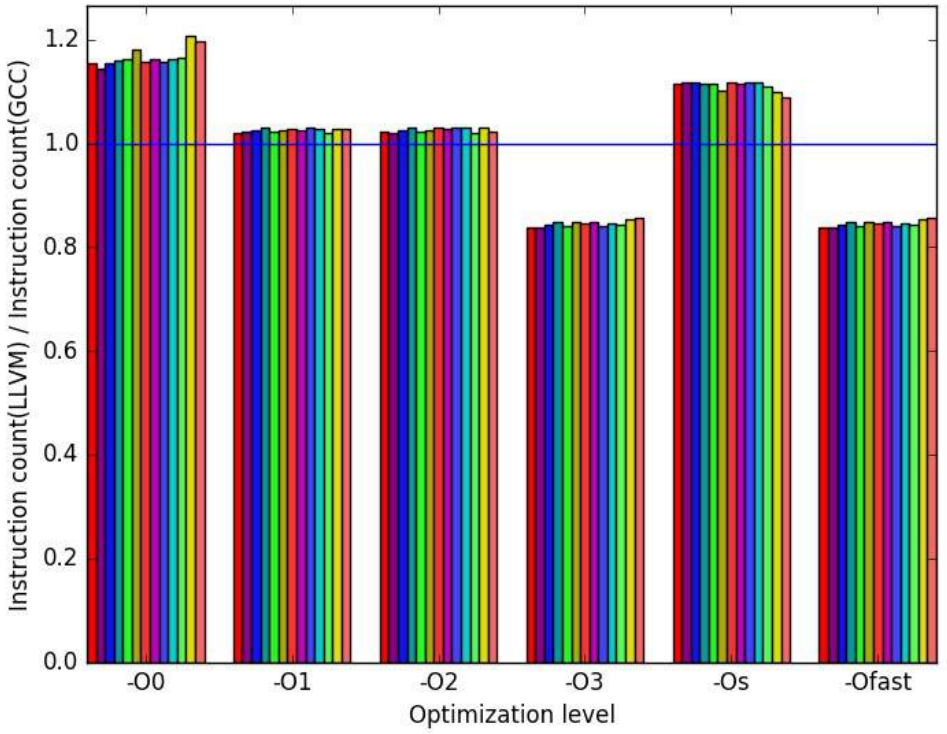
LLVM Optimization Flags

- O0: no optimization
  - Compiles the fastest and generates the most debuggable code
- O1: somewhere between O0 and O2
- O2: moderate level of optimization enabling most optimizations
- O3: like O2,
  - except that it enables opts that take longer to perform or that may generate larger code (in an attempt to make the program run faster)
- Os: like O2 with exta opts to reduce code size
- Oz: like Os, but reduce code size further
- O4: enables link-time opt Clang has support for O4, but not opt

Performance at Varying Flags

- Compare the performance of the benchmark when compiled with either GCC or LLVM
  - Compile benchmark at six optimization levels
  - Each workload was run 3 times with each executable on the Intel Core i7-2600 machines

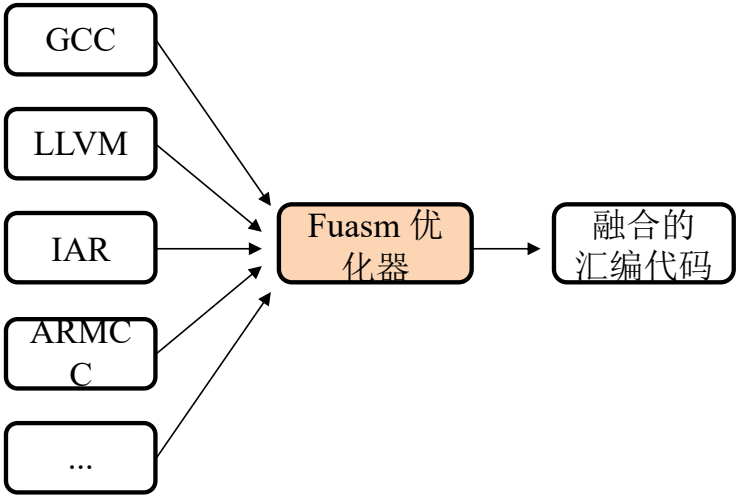


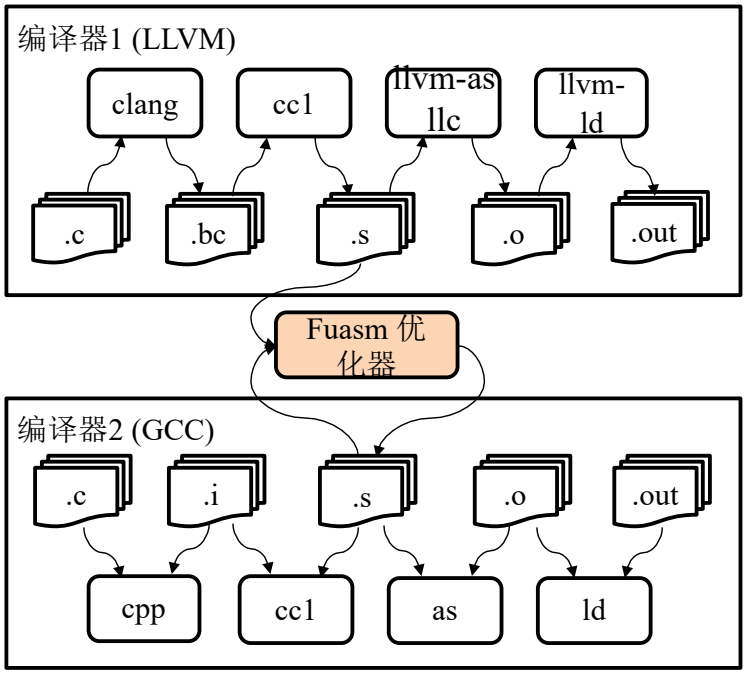


https://webdocs.cs.ualberta.ca/~amaral/AlbertaWorkloadsForSPECCPU2017/reports/exchange2_report.html#x1-12003r1

Combine GCC/LLVM?

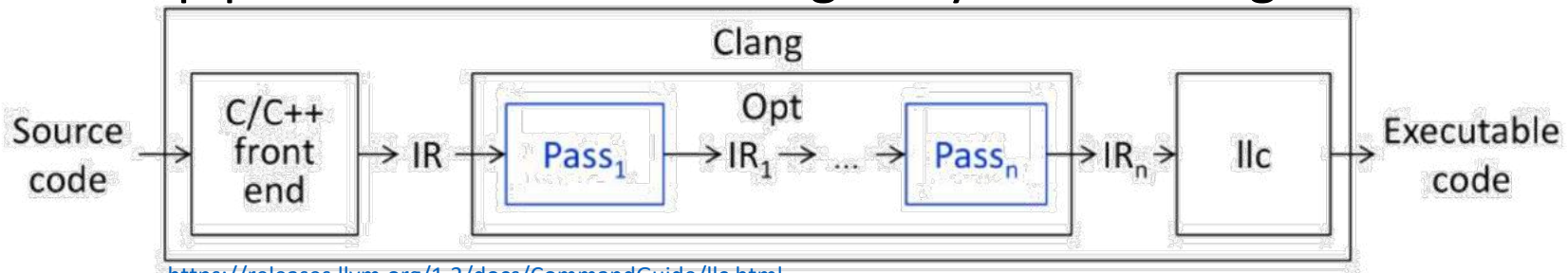
- 目标同一段代码在不同的编译器上存在性能差异
  - 不同编译器的优化策略不同
  - 不同编译器的优化方法实现不同
- 多编译器性能优化
  - Fuasm:编译基于函数替换的汇编代码融合





LLVM Passes

- Optimizations are implemented as **Passes** that traverse some portion of a program to either collect information or transform the program
- A Pass receives an LLVM IR and performs analyses and/or transformations
  - Using opt, it is possible to run each Pass
- A Pass can be executed in a middle of compiling process from source code to binary code
  - The pipeline of Passes is arranged by Pass Manager



https://releases.llvm.org/1.2/docs/CommandGuide/llc.html

The llc command compiles LLVM bytecode into assembly language for a specified architecture.

The assembly language output can then be passed through a native assembler and linker to generate native code.

LLVM Passes (cont.)

- **Analysis** passes: compute info that other passes can use or for debugging or program visualization purposes
  - -memdep: Memory Dependence Analysis (https://llvm.org/doxygen/MemDepPrinter_8cpp_source.html)
  - -instcount: Counts the various types of Instructions (https://llvm.org/doxygen/InstCount_8cpp_source.html)
  - … (https://llvm.org/doxygen/dir_a25db018342d3ae6c7e6779086c18378.html)
- **Transform** passes: can use (or invalidate) the analysis passes, all mutating the program in some way
  - -dce: Dead Code Elimination (https://llvm.org/doxygen/DCE_8cpp_source.html)
  - -loop-unroll: Unroll loops (https://llvm.org/doxygen/LoopUnrollPass_8cpp_source.html)
  - … (https://llvm.org/doxygen/dir_a72932e0778af2811509546 8f6286ff8.html)
- **Utility** passes: provides some utility but don't otherwise fit categorization
  - -view-cfg: View CFG of function
  - …

Example

- $clang -emit-llvm -S sum.c

```
int sum(int a, int b) {
  return a + b;
}
```

- $opt sum.ll -debug-pass=Structure -mem2reg -S -o sum-O1.ll

```
Pass Arguments:  -targetlibinfo -tti -targetpassconfig -assumption-cache-tracker -domtree -mem2reg -verify -print-module
Target Library Information
Target Transform Information
Target Pass Configuration
Assumption Cache Tracker
  ModulePass Manager
    FunctionPass Manager
      Dominator Tree Construction
      Promote Memory to Register
      Module Verifier
    Print Module IR
```

$opt sum.ll -debug-pass=Structure -O1 -S -o sum-O1.ll
$opt sum.ll -time-passes -O1 -o sum-tim.ll

- $opt sum.ll -time-passes -mem2reg -o sum-tim.ll

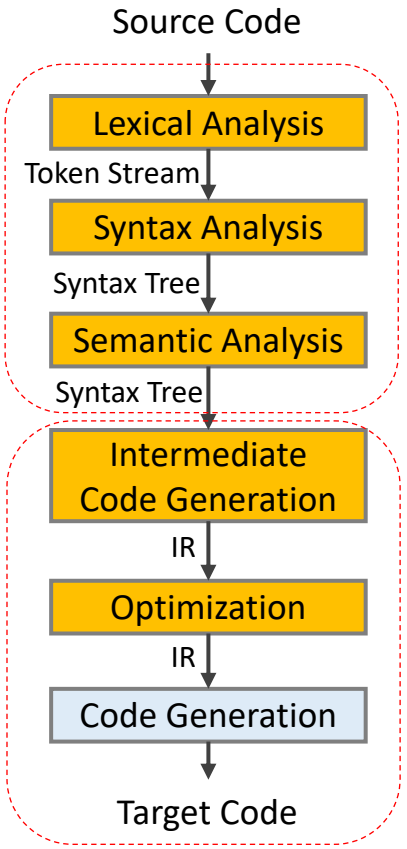
```
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0003 seconds (0.0003 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0002 ( 91.1%)   0.0001 ( 90.2%)   0.0003 ( 90.8%)   0.0003 ( 90.6%)  Bitcode Writer
   0.0000 (  3.7%)   0.0000 (  4.5%)   0.0000 (  4.0%)   0.0000 (  3.7%)  Module Verifier
   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.8%)  Dominator Tree Construction
   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.4%)  Promote Memory to Register
   0.0000 (  0.5%)   0.0000 (  0.8%)   0.0000 (  0.6%)   0.0000 (  0.6%)  Assumption Cache Tracker
   0.0002 (100.0%)   0.0001 (100.0%)   0.0003 (100.0%)   0.0003 (100.0%)  Total

===-------------------------------------------------------------------------===
                                LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0006 seconds (0.0006 wall clock)
```

Target Code Generation[目标代码生成]

- What we have now
  - Optimized IR of the source program
    - And, symbol table
- Target code
  - Binary (machine) code
  - Assembly code
- Goals of target code generation
  - Correctness: the target program must preserve the semantic meaning of the source program
  - High-quality: the target program must make effective use of the available resources of the target machine
  - Fast: the code generator itself must runs efficiently

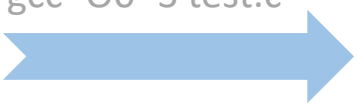


Example

- An example on real machine (*x86_64*)
  - Symbols have to be translated to memory addresses

```
1 int x = 1;
2 int y = 2;
3 int z = 3;
4
5 void main() {
6   x = y + z;
7 }
```

gcc -O0 -S test.c



```
movl    _y(%rip), %eax
addl    _z(%rip), %eax
movl    %eax, _x(%rip)
```

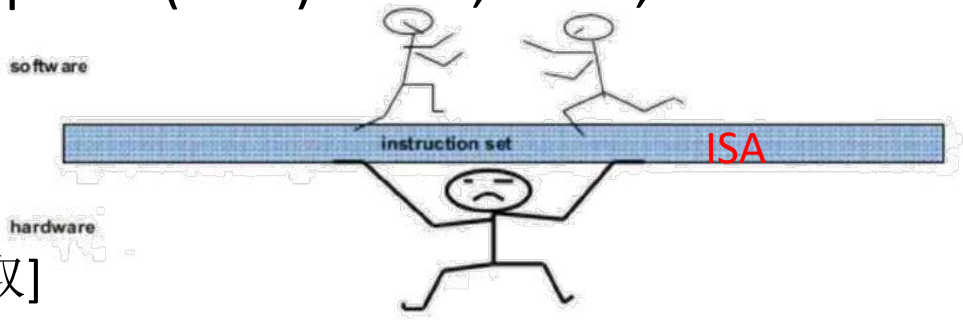
- A simplified representation

x = y + z

```
LD R0, y          // R0 = y (load y into register R0)
ADD R0, R0, z     // R0 = R0 + z (add z to R0)
ST x, R0          // x = R0 (store R0 into x)
```

Translating IR to Machine Code[翻译]

- Machine code generation is machine ISA dependent*
  - Complex instruction set computer (CISC): x86
  - Reduced instruction set computer (RISC): ARM, MIPS, RISC-V

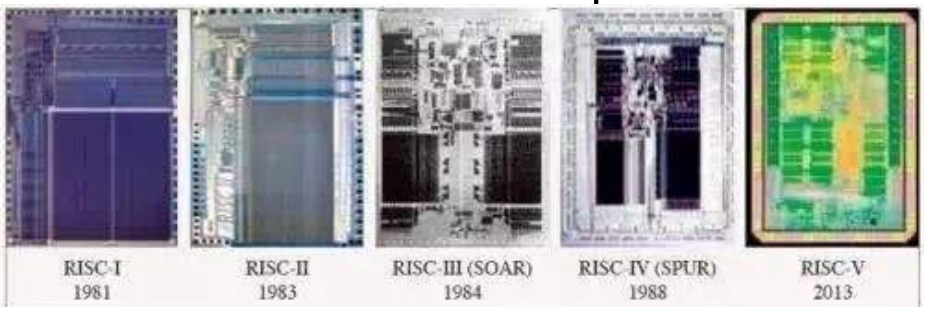


- Three primary tasks
  - Instruction selection[指令选取]
    - Choose appropriate target-machine instructions to implement the IR statements
  - Register allocation and assignment[寄存器分配]
    - Decide what values to keep in which registers
  - Instruction ordering[指令排序]
    - Decide in what order to schedule the execution of instructions

* CPU及指令集演进 (漫画 | 20多年了，为什么国产CPU还是不行？)

x86 → ARM → RISC-V[进行中的变革]

- The war started in mid 1980's
  - CISC won the high-end commercial war (1990s to today)
  - RISC won the embedded computing war
- But now, things are changing …
  - Fugaku: ARM-based supercomputer, Apple ARM-based M1 chip
- RISC-V: a freely licensed open standard (Linux in hw)
  - Builds on 30 years of experience with RISC architecture, "cleans up" most of the short-term inclusions and omissions
    - Leading to an arch that is easier and more efficient to implement



https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html
The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC

Instruction Selection[指令选取]

- Code generation is to map the IR program into a code sequence that can be executed by the target machine[选择适当的目标机器指令来实现IR]
  - ISA of the target machine
    - If there is 'INC', then for a = a + 1, 'INC a' is better than 'LD a; ADD a, 1'
  - Desired quality of the generated code
    - Many different generations, naïve translation is usually correct but very inefficient

TAC code:

```
a = b + c
d = a + e
```

Target code:

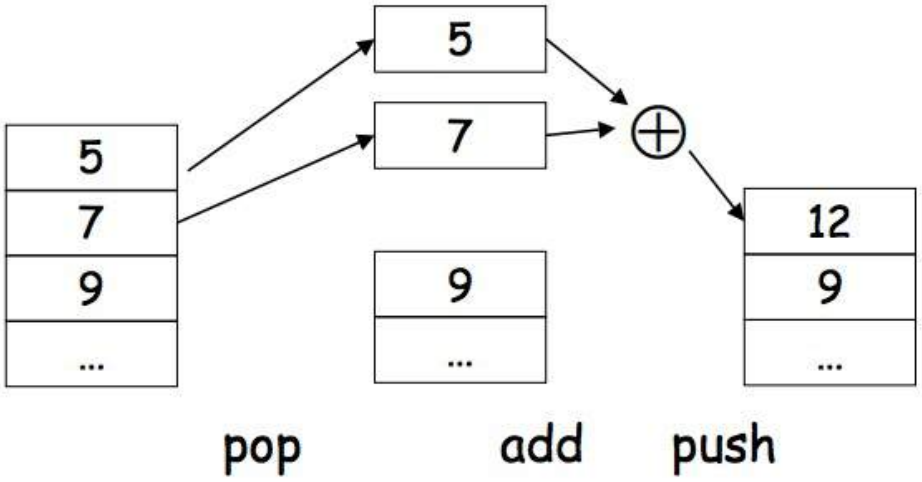
```
LD R0, b          // R0 = b
ADD R0, R0, c     // R0 = R0 + c
ST a, R0          // a = R0
LD R0, a          // R0 = a
ADD R0, R0, e     // R0 = R0 + e
ST d, R0          // d = R0
```

Register Allocation & Evaluation Order

- **Register allocation**: a key problem in code generation is deciding what values to hold in what registers[寄存器分配]
  - Registers are the fastest storage unit but are of limited numbers
    - Values not held in registers need to reside in memory
    - Insts involving register operands are much shorter and faster
  - Finding an optimal assignment of registers to variables is NP-hard
- **Evaluation order**: the order in which computations are performed can affect the efficiency of the target code[执行顺序]
  - Some computation orders require fewer registers to hold intermediate results than others
  - However, picking a best order in the general case is NP-hard

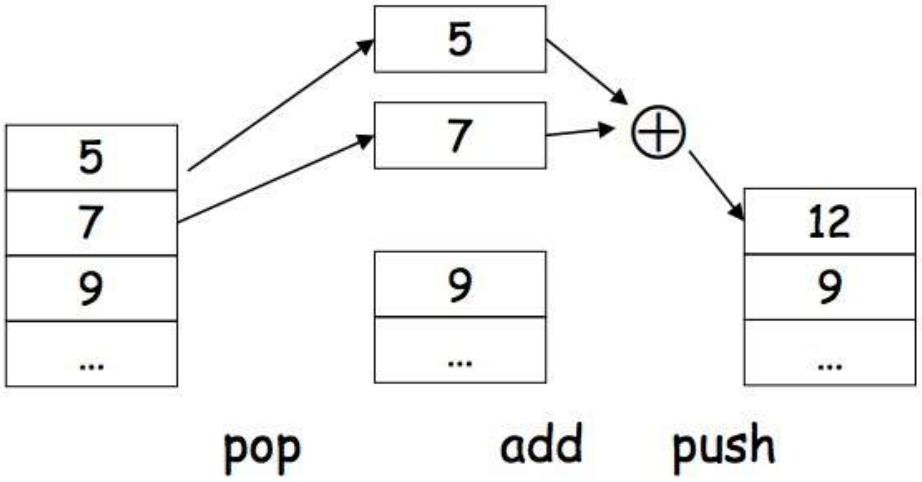
Stack Machine[栈式计算机]

- A simple evaluation model[一个简单模型]
  - No variables or registers
  - A stack of values for intermediate results
- Each instruction[指令任务]
  - Takes its operands from the top of the stack[栈顶取操作数]
  - Removes those operands from the stack[从栈中移除操作数]
  - Computes the required operation on them[计算]
  - Pushes the result on the stack[将计算结果入栈]



Example

- Consider two instructions
  - *push i* - place the integer *i* on top of the stack
  - *add* - pop two elements, add them and put the result back on the stack
- A program to compute *7 + 5*
  - *push 7*
  - *push 5*
  - *add*

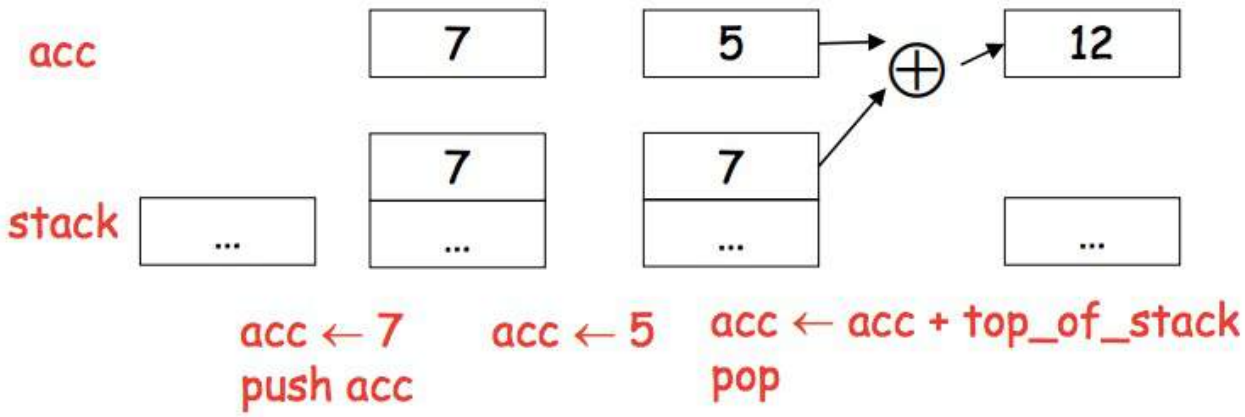


Optimize the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed

```
push 7
push 5
add
```

- **Idea**: keep the top of the stack in a register (called *accumulator*)[使用寄存器]
  - Register accesses are much faster
- The "add" instruction is now
  - *acc* ← *acc* + *top_of_stack*
  - Only one memory operation



From Stack Machine to MIPS

- The compiler generates code for a stack machine with accumulator
  - The accumulator is kept in MIPS register *$t0*
  - Stack machine instructions are implemented using MIPS instructions and registers
  - We want to run the resulting code on the MIPS processor (or simulator)
- The stack is kept in memory
  - The stack grows towards lower addresses (standard convention)
  - The address of next stack location is kept in a MIPS register *$sp*
    - The top of the stack is now at address *$sp + 4*
  - A block of stack space, called **stack frame**, is allocated for each function call
    - A stack frame consists of the memory between $fp which points to the base of the current stack frame, and the $sp
    - Before func returns, it must pop its stack frame, and restore the stack

MIPS Architecture

- Load/store architecture
  - Only load and store instructions can access memory
  - All other instructions access only registers
    - E.g., all arithmetic and logical operations involve only registers (or constants that are stored as part of the instructions)
- Word size is 32 bits, all instructions are encoded in a single 32-bit word format
  - Arithmetic
    - e.g., add des, src1, src2 // des = src1 + src2
  - Comparison
    - e.g., sge des, src1, src2 // des ← 1 if src1 ≥ src2, 0 ow
  - Branch/jump
    - e.g., bge src1, src2, lab // branch to lab if src1 ≥ src2
  - Load, store, and data movement
    - E.g., lw des, addr // load the word at addr into des
    - E.g., move des, src1 // copy the contents of src1 to des

MIPS Architecture (cont.)

- 32 registers
  - 31 of these are general-purpose that can be used in any of the instructions
  - The last one (*zero*), is to contain the number zero at all times
- While general-purpose, there are guidelines specifying how each of the registers should be used
  - $0 is always zero, $a0,...,$a4 are for arguments
  - $sp saves stack pointer, $fp saves frame pointer

Symbolic Name	Number	Usage
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers 1 $\cdots$ 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 $\cdots$ 9.
s0 - s7	16 - 23	Saved Registers 0 $\cdots$ 7.
k0 - k1	26 - 27	Kernel Registers 0 $\cdots$ 1.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

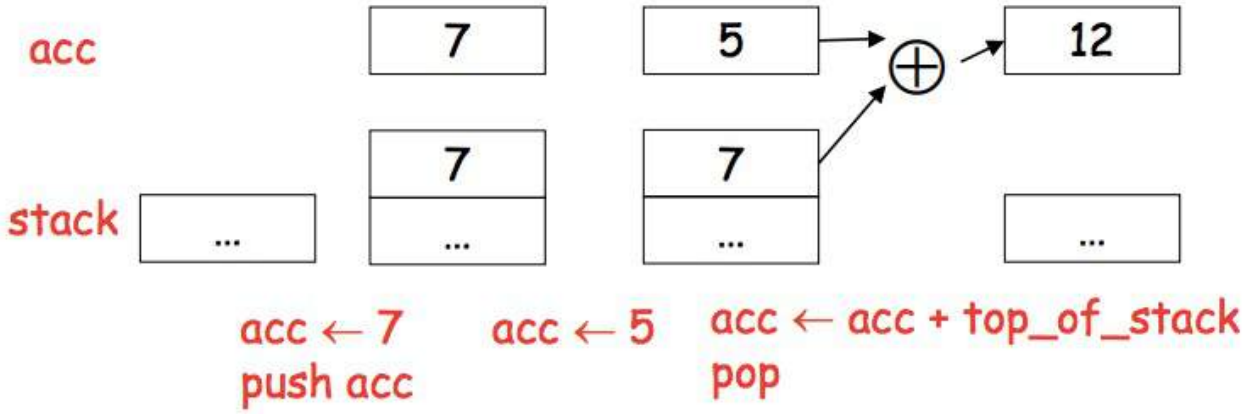
Example MIPS Instructions

- *la reg1 addr*
  - Load address into reg1
- *li reg imm*
  - reg ← imm
- *lw reg1 offset(reg2)*
  - Load 32-bit word from address reg2 + offset into reg1
- *sw reg1 offset(reg2)*
  - Store 32-bit word in reg1 at address reg2 + offset
- *add reg1 reg2 reg3*
  - reg1 ← reg2 + reg3
- *move reg1 reg2*
  - reg1 <- reg2
- *sge reg1 reg2 reg3*
  - reg1 ← (reg2 >= reg3)

Example MIPS Assembly

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

Stack-machine	**MIPS**	**Comment**
acc <- 7	li $t0 7	Load constant 7 into $t0
push acc	addi $sp $sp -4 sw $t0 0($sp)	Decrement sp to make space Copy the value to stack
acc <- 5	li $t0 5	Load constant 5 into $t0
acc <- acc + top_of_stack	lw $t1 4($sp) add $t0 $t0 $t1	Load value from $sp+4 into $t1 Add $t0+$t1 = 5 + 7
pop	add $sp $sp 4	Pop constant 7 off stack



A Small Language

- A language with integers and integer operations

$P \rightarrow D; P \mid D$
$D \rightarrow \text{def id}(ARGS) = E;$
$ARGS \rightarrow \text{id}, ARGS \mid \text{id}$
$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$
$\quad \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$

- Example: program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
                    if x = 2 then 1 else
                      fib(x - 1) + fib(x – 2)
```

Code Generation Considerations[考虑]

- We used to store values in unlimited temporary variables, but registers are limited --> must reuse registers[重复使用寄存器]
- Must save/restore registers when reusing them[保存-恢复]
  - E.g. suppose you store results of expressions in $t0
  - When generating $E \rightarrow E_1 + E_2$,
    - E_1 will first store result into $t0
    - E_2 will next store result into $t0, overwriting E_1's result
    - Must save $t0 somewhere before generating E_2
- Registers are saved on and restored from the stack

  Note: $sp - stack pointer register, pointing to the top of stack
  - Saving a register $t0 on the stack:

```
addiu $sp, $sp, -4     # Allocate (push) a word on the stack
sw $t0, 0($sp)         # Store $t0 on the top of the stack
```

  - Restoring a value from stack to register $t0:

```
lw $t0, 0($sp)         # Load word from top of stack to $t0
addiu $sp, $sp, 4      # Free (pop) word from stack
```

Stack Operations[栈操作]

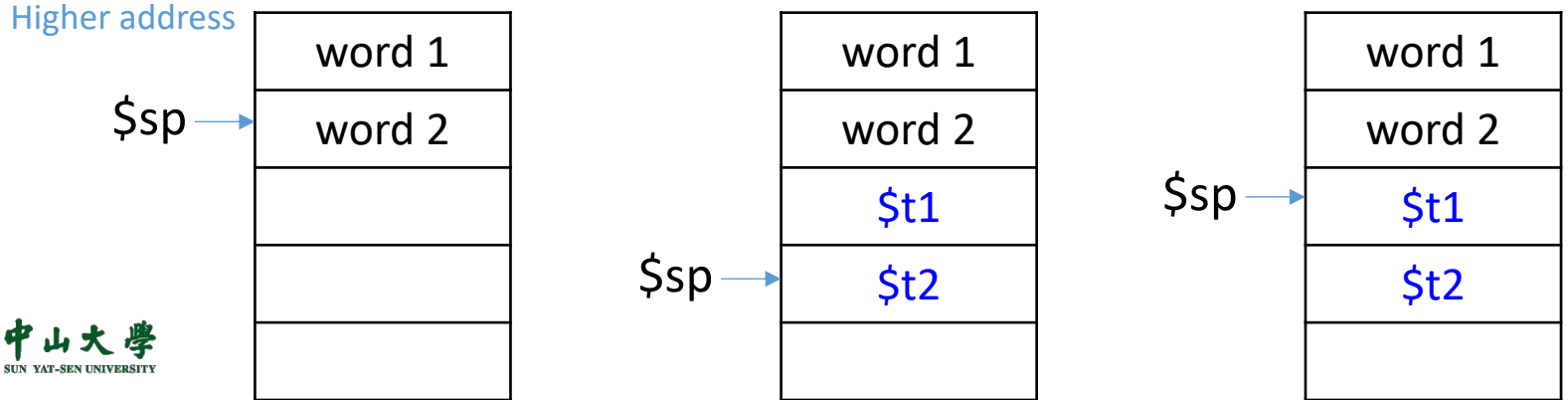
- To **push** elements onto the stack
  - To move stack pointer $sp down to make room for the new data
  - Store the elements into the stack
- For example, to push registers $t1 and $t2 onto stack

```
sub $sp, $sp, 8
sw $t1, 4($sp)
sw $t2, 0($sp)
```

⟺

```
sw $t1, -4($sp)
sw $t2, -8($sp)
sub $sp, $sp, 8
```

- **Pop** elements simply by adjusting the $sp upwards
  - Note that the popped data is still present in memory, but data past the stack pointer is considered invalid



Code Generation Strategy

- For each expression *e* we generate MIPS code that:
  - Computes the value of *e* into *$t0*
  - Preserves *$sp* and the contents of the stack
- We define a code generation function *cgen(e)*
  - Its result is the code generated for *e*
- Code generation for constants
  - The code to evaluate a constant simply copies it into the register: *cgen(i) = li $t0 i*
    - Note that this also preserves the stack, as required