Code Generation: x64 Assembly

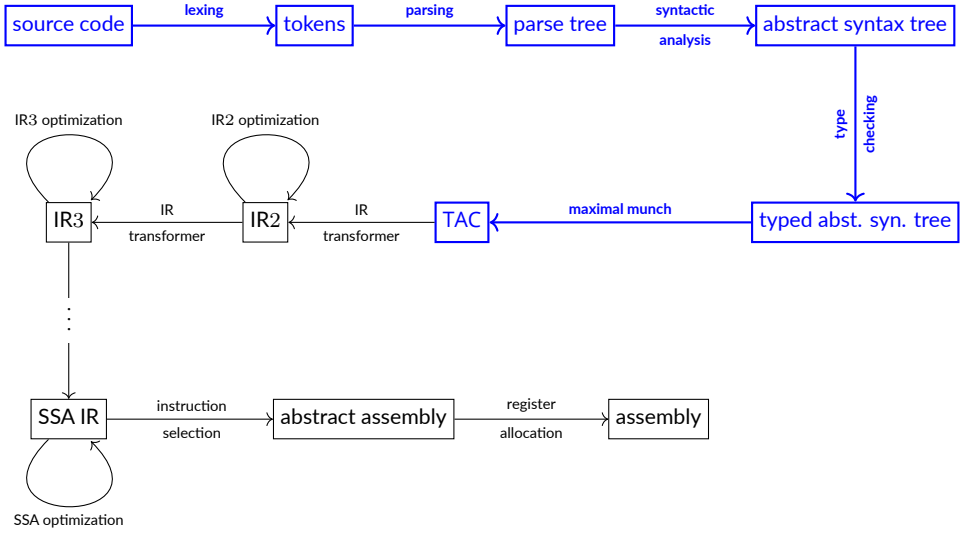
(CSC 3F002 EP) Compilers

Pierre-Yves Strub

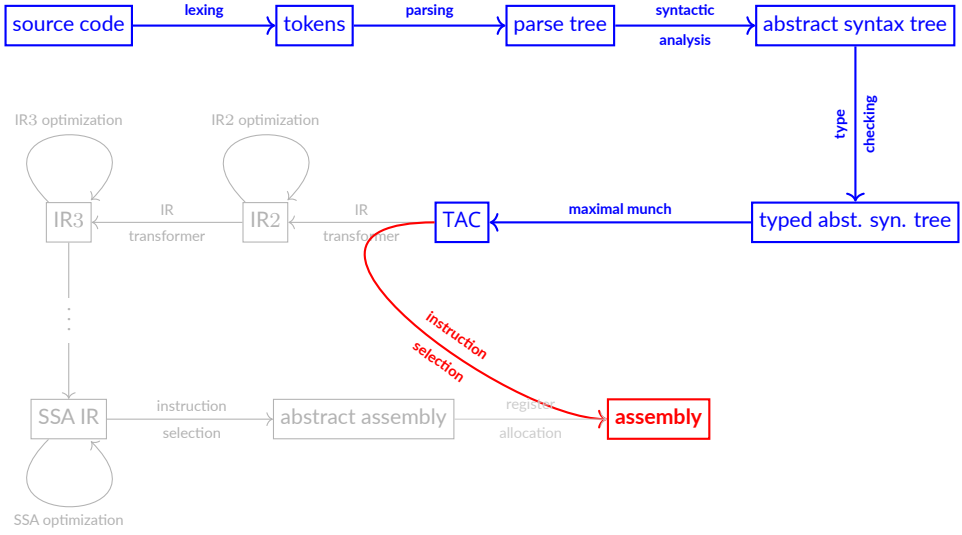
(Slide deck author: **Kaustuv Chaudhuri**)

2025-09-22 // Week 04

What You've Written So Far



Today's Focus



Agenda for Today

- A brief introduction to x64
  - Temporaries and the stack
  - Arithmetic and Bitwise instructions
  - Comparisons and Jumping

- Going from TAC to x64 — lab 3, week 2

Brief Introduction to x64

Assembly Program: Structure

- Three main kinds of sections, all *fixed size*:
  - Text: contains *non-modifiable* instructions (*aka: code*)
  - Data: contains *modifiable*, *initialized* data
  - BSS: contains *modifiable*, *uninitialized* (0-filled) data
    ("Block Started by Symbol", "Better Save Space")
  - Read-Only Data (rodata): non-modifiable initialized data, usually considered part of the text section
- In an assembly file these sections can be interleaved in an arbitrary order, but assemblers usually de-interleave them
- Every section declares a list of labels of two varieties
  - Local: not exported by object code
  - Global: exported by object code
- In text segments, labels are jump addresses
- In data or bss segments, labels are pointers

Assembly Program: Structure Example

```
# Line comments begin with '#'
/* Can also use C/C++-style block comments */

    ## declare all the global symbols

    .globl main         # note: not .glob*a*l
    .text
main:                   # note: globally accessible function
    ## instructions for main function

fib:                    # note: known only within this file because no .globl
    ## instructions for fib function

    .section .rodata
msg:                    # note: known only within this file because no .globl
    ## e.g., declaration of a string (i.e., byte array) named 'msg'

    .globl values
    .section .data
values:                 # note: globally accessible
    ## e.g., declaration of an array named 'values'
```

A Full(y Trivial) Example

(File `ret42.s`)

```
## A program that only returns code 42
    .globl main
    .text
main:
    movq $42, %rax
    retq
```

```
$ as --64 ret42.s -o ret42.o          # compile ret42.s to ret42.o

$ gcc -o ret42.exe ret42.o            # link ret42.o with the C runtime
                                      # to create the binary ret42
$ ./ret42.exe

$ echo $?                             # print exit code of prev. command
42
```

Disassembly

```
$ as --64 ret42.s -o ret42.o            # assemble into object file

$ objdump -d ret42.o                    # disassemble object file

ret42.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   48 c7 c0 2a 00 00 00    mov    $0x2a,%rax
   7:   c3                      retq

$ objdump --disassemble=main ret42.o    # can also pick function to disassemble
 ... same output as above ...
```

Stepping Through Assembly

- Compile the code with `gcc` using the `-g` flag

```
$ gcc -g -o ret42.exe ret42.s       # one shot compilation
```

- Use `gdb`, the system debugger

```
$ gdb ret42.exe
... bunch of output ...
Reading symbols from ret42.exe...
(gdb) l                                                 shows the assembler source
1        .globl main
2        .text
3    main:
4        movq $42, %rax
5        retq
(gdb) b 5                                               set breakpoint on line 4
Breakpoint 1 at 0x7: file ret42.s, line 5.
(gdb) r                                                 begin execution
Starting program: /home/kaustuv/.../ret42.exe

Breakpoint 1, main () at ret42.s:5
5        retq
(gdb) i r rax                                           display contents of RAX
rax            0x2a                42
(gdb) c                                                 continue
Continuing.
[Inferior 1 (process 105074) exited with code 052]
(gdb)
```

Computer Memory: Quick Recap

- Memory: stores data, addressable
  - Volatile memory: contents invalidated on power loss
  - Persistent memory: contents preserved on power loss
    (aka *non-volatile* memory)
- A computer has many kinds of memory
  - Storage: hard disks, flash memory, etc. (*persistent*)
    - Seek latency: spinning disk: 3ms, SSD: 16μs
  - Main Memory: memory modules, variable (*volatile*)
    - Reference: 100ns
  - Cache: fixed, part of the processor die (*volatile*)
    - L2 reference: 4ns, L1 reference: 0.5ns
  - Registers: fixed, hard-wired in the processor logic (*volatile*)
    - Latency: depends on clock speed. E.g., 4GHz proc. has ≤ 0.25ns
    - Fully parallelized: every bit accessed in parallel

link: excellent interactive visualization tool for latency

x64 Registers

- General purpose registers (GPR) – stores integers
  `RAX RBX RCX RDX RSI RBP RSP RDI`
  `R8  R9  R10 R11 R12 R13 R14 R15`
- Floating point registers (FPR) – stores floats
  `FPR0 FPR1 FPR2 ··· FPR7`
- Special purpose registers (SPR)
  `RIP RFLAGS`
- MMX registers (64 bits) – SSE
  `MMX0 MMX1 MMX2 ··· MMX7`
- XMM registers (128 bits) – SSE2
  `XMM0 XMM1 XMM2 ··· XMM7`
- YMM registers (256 bits) – SSE3, SSSE3
  `YMM0 YMM1 YMM2 ··· YMM7`
- ZMM registers (512 bits; not yet consumer-grade) – AVX-512
  `ZMM0 ZMM1 ZMM2 ··· ZMM7`

GPRs and the C Calling Convention (ABI)

- Some of the GPRs have special interpretations
- Stack frame:
  - `RBP`: *base pointer*, points to *end* of allocated stack and start of book-keeping data
  - `RSP`: *stack pointer*, points to *start* of allocated stack
  - Don't use these two registers for computations!
- Callee-save registers: ("*non-volatile*")
  - Must retain values across function calls
  - In the C (SysV) ABI: `RBX`, `RBP`, `R12`, `R13`, `R14`, `R15`
  - Don't use any of these for lab 3 – we'll come back to them later
- Caller-save register: ("*volatile*")
  - Any GPR that is not callee-save
  - Callers must keep copies if the old values matter to them
  - Called procedures can overwrite them...
    ... so, feel free to use them for lab 3

Move Immediate

```
movq     $42,   %rax
       immediate destination
```

- Stores a given number (“*immediate*”, prefixed `$`) in a register
- Note: the immediate must fit in 32 bits!
- Immediate value can be given in Hexadecimal (e.g., `$0x42`)

Move (Copy) Register

```
movq      %rax, %rdx
          source  destination
```

- Both source and destination register must be the same size
- If source and destination are the same, this is a null operation

Addition and Subtraction (Registers)

```
addq    %rdx, %rax
        source  destination
```

- Adds the value of the source register to the value of the destination register (think: `%rax = %rax + %rdx`)
- Overflow flags are set based on destination (∼ *lab 3*, *checkpoint 2*)
- Subtraction (`subq`) defined similarly.

Addition and Subtraction (Immediates)

```
addq     $42,   %rax
```
immediate destination

- Adds the immediate to the value of the destination register (think: `%rax = %rax + 42`)
- Subtraction (`subq`) defined similarly.

Arithmetic Negation

```
negq    %rax
        source/dest
```

- Negates the value of the source register and stores it back in the same register (think: `%rax = - %rax`)

Signed Multiplication

```
imulq     $42,    %rax
          factor  destination
```

- Multiplies the value of the factor to the destination register.
- There is a more complicated single operand version of `imulq` that has much higher precision, but you may choose to ignore it for this course.

Signed Division and Modulus

```
idivq    %rcx
         divisor
```
(3 bytes)

- Divides the value of `RDX:RAX` by the value of the divisor register, and stores the quotient in `RAX` and remainder (modulus) in `RDX`
- Think:

```
%rax = (%rdx:%rax) / %rcx
%rdx = (%rdx:%rax) % %rcx
```

- Because this can obliterate `RDX`, you may want to save and restore its value using a book-keeping register (e.g., `R11`)
- To sign-extend a value stored in `RAX` to `RDX:RAX`, use the nullary instruction: `cqto`.

Shifting Left

(shifting right is similar)

```
salq    $7,    %rax
          count   destination

salq    %cl,   %rax
          count   destination
```

- Shifts the destination register left by the count specified as an immediate (which must fit in 6 bits) or in the register `CL` (which is the low 8 bits of `RCX`) (Think: `%rax = %rax << 7`)
- Note: no other register besides `CL` can be used as the count

Bitwise And, Or, Xor

```
andq    $42,    %rax
        imm. src.   destination

andq    %rdx,   %rax
        reg. src.   destination
```

- Computes the bitwise and of the source (either an immediate or the value of a register) and the value of the destination register, and stores the result in the destination register. (Think: `%rax = %rax & %rdx`, etc.)
- Similarly for or (`orq`) and xor (`xorq`)

Bitwise Complement

```
notq     %rax
         source/dest
```

- Takes the bitwise complement (aka 1's complement) of the value of the source register and stores it back in the same register
  (Think: `%rax = ~ %rax`)

Other Instructions

- The collection of instructions presented so far will suffice for all arithmetic and bitwise operations with registers and immediates in this course
- However, you may wish to use some additional instructions that are optimized for specific use cases. Some suggestions:
  - `inc` and `dec`: increment and decrement by 1
  - `movsx` and `movsxd`: sign extend to/from arbitrary registers, not just `RAX`
  - `movzx`: move with *zero*-extension
  - `andn`: "and not" (Haswell microarchitecture and later)
  - `rol`, `ror`: rotate left/right

Labels and Instruction Pointer

- Pointers: (virtual) memory addresses as values
  Can be dereferenced to read/write memory
- Label: addresses of an instruction in compiled code
- Instruction pointer (`RIP`): contains the address of the *next* instruction, so setting `RIP` changes the code path
- Jump: specialized instructions to set `RIP`.
  - `jmp`: absolute jump
  - `jz`, `jnz`, `jl`, `jle`, etc: conditional jumps

Comparison Instruction

```
cmpq      $42,   %rax
          src2   src1
       (reg/imm) (reg/mem)
```

- Sets the status flags register (`%eflags`) based on src1 − src2
- Note: does not affect any of the other registers
- Overlays:
  - `cmpl  $42, %eax`
  - `cmpw  $42, %ax`
  - `cmpb  $42, %al`

Jump Instructions

```
jmp     .L42
jcnd    .L42
        label
```

- Jumps unconditionally (`jmp`) or when a particular condition flag is set/unset based on an earlier `cmpq`
- Some conditional jumps: (note synonyms)

Instruction	Condition	wrt: [`cmpq src2, src1`]
`je`, `jz`	ZF=1	`src1 - src2 == 0`
`jne`, `jnz`	ZF=0	`src1 - src2 != 0`
`jle`, `jng`	ZF=1 or SF≠OF	`src1 - src2 <= 0`
`jl`, `jnge`	ZF=0 and SF≠OF	`src1 - src2 < 0`
`jge`, `jnl`	ZF=1 or SF≠OF	`src1 - src2 >= 0`
`jg`, `jnle`	ZF=0 and SF=OF	`src1 - src2 > 0`

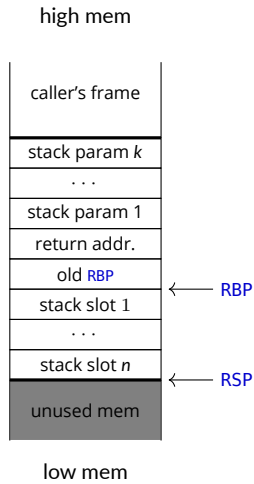
Storing Temporaries in the Stack

- There are a finite number (≤ 14) of usable registers.
  What if we have more variables/intermediates/temporaries?

- Answer: we *spill* some temporaries onto the stack
- General spilling algorithm:
  1. Assign one register (say `R11`) for book-keeping spilled temps
  2. Allocate a *slot* on the stack for each temporary that is to be spilled.
  3. Whenever that temporary is needed, load it from the stack slot into `R11`, perform the operation, then save `R11` back into the stack slot.
- Sometimes the stack slot can itself be an operand (stay tuned!)

The Stack Frame, Briefly

(we'll cover the stack in full gory detail in lab 4)



Using the Stack

- Until lab 4 (procedures), we will ignore:
  - Stack parameters
  - Return address
- All we need for now: there are *n* stack slots between `RSP` and `RBP` available to be used for temporaries.
- What is *n*?
  - First approximation: number of temporaries in TAC
  - Better: temporaries can be allocated to caller-save registers that are not used for anything else (yet): `RSI`, `RDI`, `R8` – `R11`.
  - Warning! `RBX`, `R12` – `R15` are callee-save!

Allocating the Stack

(Hint: use this as a template for lab 3)

```
1       .globl main
2       .text
3   main:
4       # store old RBP at top of the stack
5       pushq %rbp
6       # make RBP point to just after (i.e., top) stack slots
7       movq %rsp, %rbp
8
9       # Now we allocate stack slots in units of **16** bytes (= 128 bits)
10      # E.g., for 7 slots, we would allocated 8 * 8 = 64 bytes
11      subq $64, %rsp
12
13      #
14      # The rest of the compiled code from BX1
15      #
16
17      # restore RSP and RBP
18      movq %rbp, %rsp
19      popq %rbp
20
21      movq $0, %rax       # return code 0
22      retq
```

Loading and Storing to Stack Slots

syntax for stack slots as operands

Stack slot 1: `-8(%rbp)`
Stack slot 2: `-16(%rbp)`
Stack slot 3: `-24(%rbp)`
⋮
Stack slot *n*: $-8n$`(%rbp)`

```
movq    -8(%rbp), %rax      (load slot into register)
        source    destination

movq    %rdx, -8(%rbp)      (save register to slot)
        source  destination
```

- Can use a dedicated register (e.g., `R11`) for load/store to stack slots followed/preceded by a register-register `movq`.
- Both source and destination cannot be stack slots!

Optimization: Using Stack Slots Directly as Operands

```
addq    -8(%rbp), %rax
          source    destination

addq    %rdx, -8(%rbp)
        source  destination
```

- A stack slot be used instead of a register operand when that register is arbitrary
- Note: not the case for `sarq`, `idivq`, etc. which will *always* use specific registers
- All operands cannot be stack slots!

Lab 3, Checkpoint 1

Lab 3 (control structures), Checkpoint 1

- Lab 3 is about compiling BX control instructions to x64, ...
- Checkpoint 2: converting TAC (+labels, jumps) to x64
  - Note: In x64, destination reg = final arg reg (usually)
    Translating TAC to x64 is not completely trivial
  - May need one or two book-keeping registers (use `R11` unless you need something else)

- Reminder: Lab 3 has a full duration of 2 weeks