CMSC 430: Introduction to Compilers

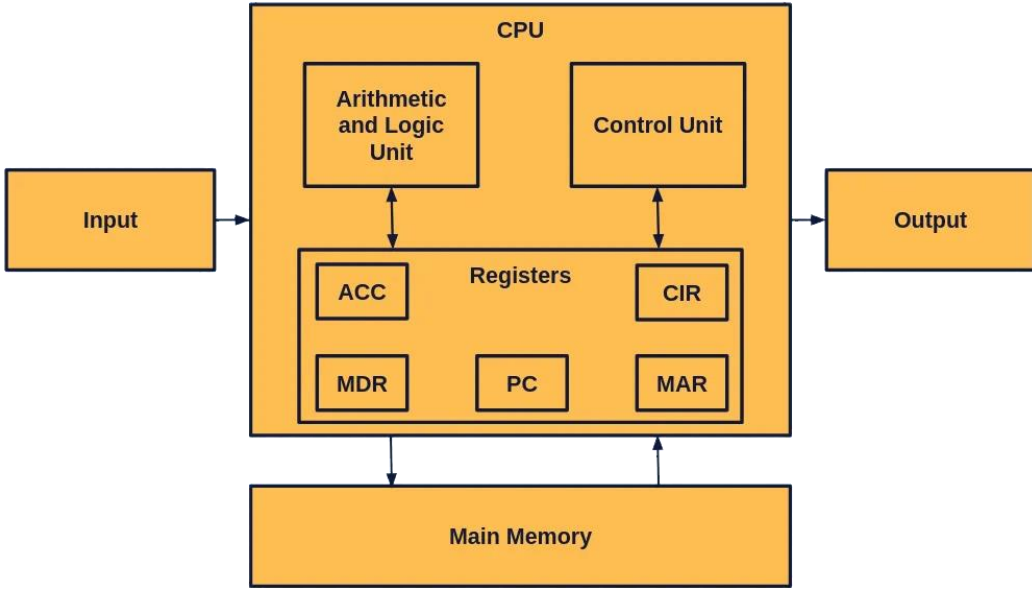
x86 and a86

Announcements 02/05/2026

- Assignment 1 is due on February 13
- Quiz 1 is due on February 10
- Lecture recordings on ELMS (under Panopto)
- Slides on ELMS (under Files)
- Course notes on webpage
- Review: Syntax error, p-dbg (print-debug/print-dbg)
- Today:
  - X86
  - A86

Von Neumann architecture

- Fetch-execute cycle:
  - PC → MAR
  - PC := PC + 8 (next instr)
  - Memory → MDR
  - MDR → CIR
  - decode and execute (CIR)
  - Repeat



x86

- x86 is an instruction set architecture (ISA)
  - a programming language whose interpreter is implemented in CPU.
- x86_64 is a 64-bit version of the x86 instruction set.
- 16 registers (64-bit)
  - rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9…,r15
- x86_64 is complicated.
  - backwards compatibility requirements

x86 Program

- Sequence of x86 instructions: label, jmp, mov, cmp
- Untyped. Only data type is 64-bit integer
- 64-bit registers and main memory

```
        global    _start
        section   .text
_start: mov rax, 0x2000004 ; write
        mov       rdi, 1     ;stdout
        lea rsi, [rel message]
        mov rdx, message_len
        syscall

        mov rax, 0x2000001 ;exit
        xor       rdi, rdi
        syscall
        section .data
message: db "Hello, world! CMSC430", 10
message_len equ $ - message
```

```
On Mac: nasm -g -f macho64 hello.s
        gcc -arch x86_64 -e _start hello.o -o hello
```

Example

```
#include <stdio.h>
#include <inttypes.h>
int64_t entry();

int main(int argc, char** argv) {
  int64_t result = entry();
  printf("%" PRId64 "\n", result);
  return 0;
}
```

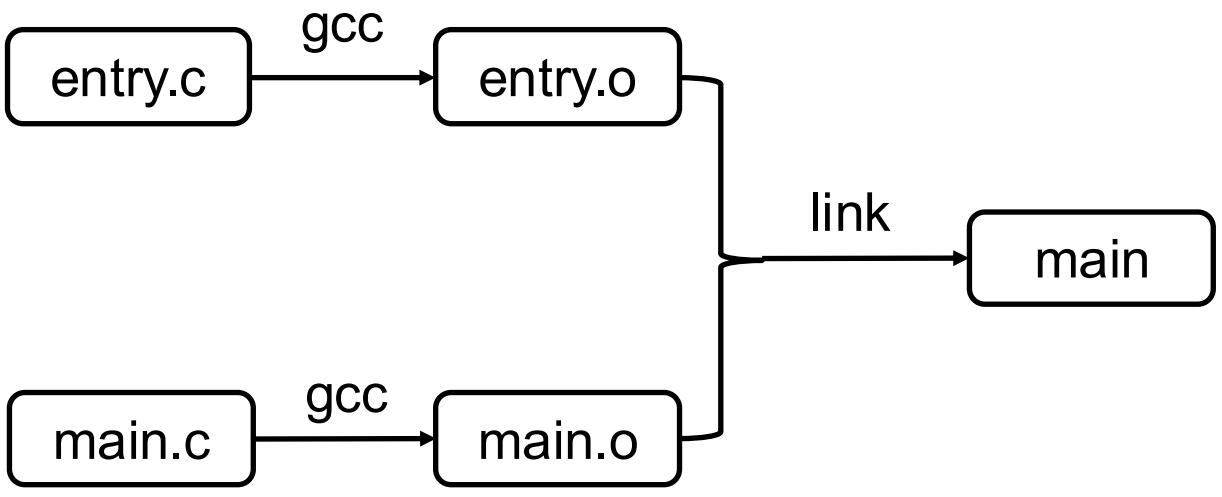
```
#include <inttypes.h>
int64_t entry(){
  return 42;
}
```

Example

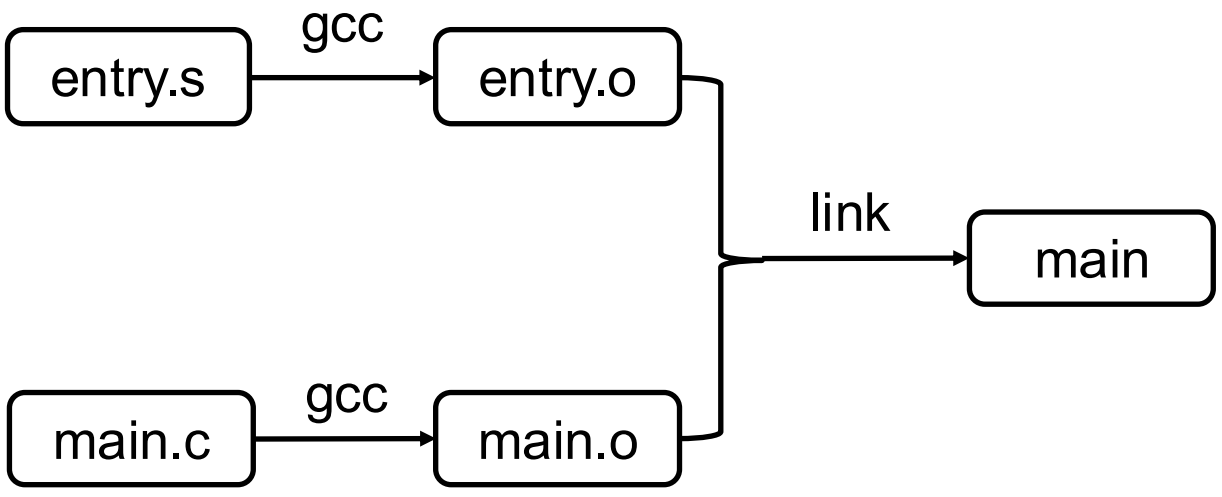
```
#include <stdio.h>
#include <inttypes.h>
int64_t entry();

int main(int argc, char** argv) {
  int64_t result = entry();
  printf("%" PRId64 "\n", result);
  return 0;
}
```

```
 .intel_syntax noprefix
        .text
        .global _entry
_entry:
        mov rax, 42
        ret
```

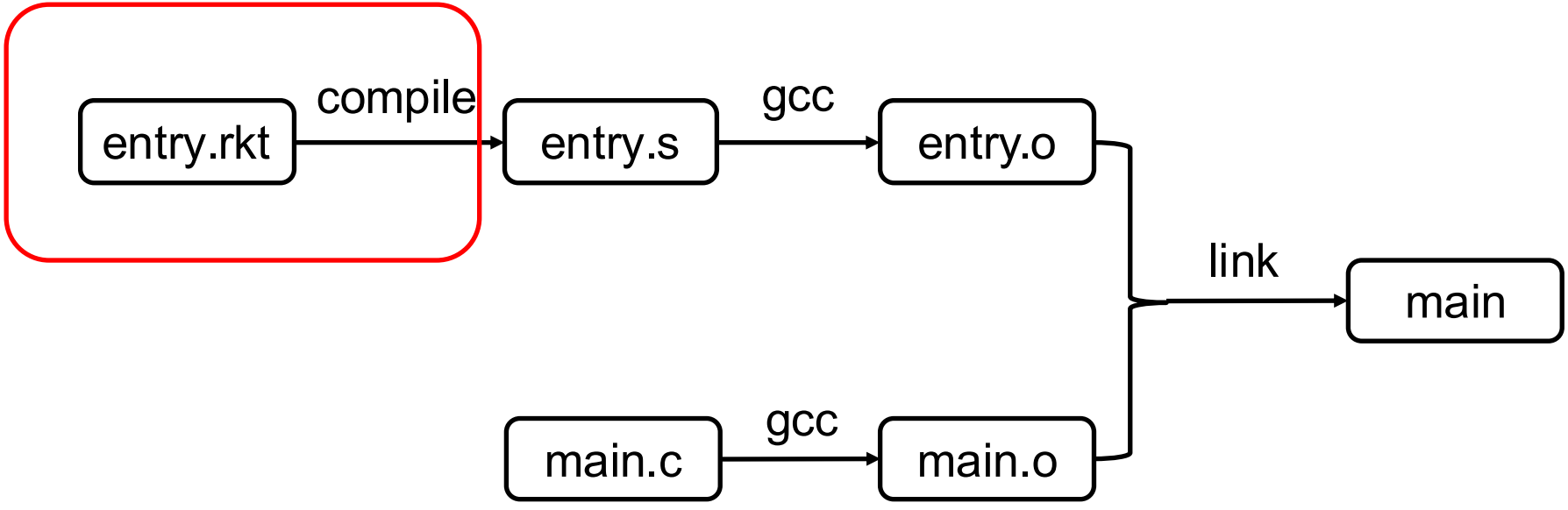
Link with the Runtime (C example)



Link with the Runtime (X86 example)



What Does Our Compiler Do?



a86: Representing x86 Code as Data

- We will represent x86 programs as data.
  - Rather than writing x86 code directly, we will instead represent them using a new data type called a86.
- An a86 program is a list of a86 instructions
  - It covers a subset of the x86 instructions.
  - Registers: 'rax, 'rbx, 'rcx, 'rdx, 'rbp, 'rsp, 'rsi, 'rdi, 'r8, 'r9, 'r10, 'r11, 'r12, 'r13, 'r14, and 'r15.
  - Each x86 instruction is represented as a structure in a86.

```
mov rax 10
```
x86

```
(Mov 'rax 10)
```
a86

Example

```
#include <stdio.h>
#include <inttypes.h>
int64_t entry();

int main(int argc, char** argv) {
  int64_t result = entry();
  printf("%" PRId64 "\n", result);
  return 0;
}
```

```
(prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rax 42)
  (Add 'rax 1)
  (Ret))
```

Some a86 Instructions

- **`Mov`**
- **`Add, Sub`**
- **`And, Or, Xor, Not`**
- **`Sal, Sar`**
- **`Label, Jmp, Call, Ret`**
- **`Push, Pop`**
- **`Cmp`**
- **`Je, Jne, Jl, Jle, Jg, Jge`**
- **`Cmov*`**

Mov

- Mov reg int64 copy integer literal into reg
- Mov reg1 reg2 register move: copy contents of reg2 into reg1
- Mov offset reg memory write: copy contents of reg into memory
- Mov reg offset memory read: copy contents of memory into reg
- Mov offset1 offset2 - illegal instruction, must go through a register
- Offset reg int64 interpret reg as a pointer and dereference at given offset

Mov

```
#lang racket
(require a86)
(asm-display
  (prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rbx 42)
  (Mov 'rax 'rbx)
  (Ret)))
```

a86

```
.intel_syntax noprefix
        .text
        .global _entry
_entry:
        mov rbx, 42
        mov rax, rbx
        ret
```

x86

Add

- Add reg int64 add integer literal into reg
- Add reg1 reg2 add reg2 into reg1

Add

```
#lang racket
(require a86)
  (prog
  (Global 'entry)
  (Label 'entry)
  (Mov `rax 32)
  (Add 'rax 10)
  (Ret))
```

Sub

```
#lang racket
(require a86)
 (prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rax 32)
  (Sub 'rax 10)
  (Ret))
```

And

```
#lang racket
(require a86)
(asm-interp
 (prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rax 9)
  (And 'rax 1)
  (Ret)))
```

```
  9: …0000 1001
  1: …0000 0001
AND: …0000 0001==>1
```

Or

```
#lang racket
(require a86)
(asm-interp
 (prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rax 9)
  (Or 'rax 3)
  (Ret)))
```

```
 9: …0000 1001
 3: …0000 0011
OR: …0000 1011 ==>11
```

Xor

```
#lang racket
(require a86)
(asm-interp
 (prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rax 9)
  (Xor 'rax 3)
  (Ret)))
```

```
  9: …0000 1001
  3: …0000 0011
XOR: …0000 1010 ==>10
```

Not

```
#lang racket
(require a86)
(asm-interp
 (prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rax 9)
  (Not 'rax)
  (Ret)))
```

```
   9: …0000 1001
NOT: …1111 0110 ==>-10
```

```
          1111 0110
Invert:   0000 1001
+1:               1
 =        0000 1010 ==>-10
```

Sar: Shift to the right

```
#lang racket
(require a86)
(asm-interp
    (prog
     (Global 'entry)
     (Label 'entry)
     (Mov 'rax 42)
     (Sar 'rax 1)
     (Ret)))
```

```
 42: …0000 0010 1010
```

Shift 1 bit to the right

```
21: …0000 0001 0101
```

If a number is encoded using two's complement, then an arithmetic right shift preserves the number's sign

```
-1: …1111 1111 1111
```

Shift 1 bit to the right

```
-1: … 1111 1111 1111
```

Sal: Shift to the left

```
#lang racket
(require a86)
(asm-interp
    (prog
     (Global 'entry)
     (Label 'entry)
     (Mov 'rax 4)
     (Sal 'rax 6)
     (Ret)))
```

```
  4: …0000 0000 0100
```

Shift 6 bits to the left

```
256: …0001 0000 0000
```

Jmp

```
#lang racket
(require a86)
(asm-interp
   (prog
    (Global 'entry)
    (Label 'entry)
    (Mov 'rax 42)
    (Jmp 'l1)
    (Mov 'rax 0)
    (Label 'l1)
    (Ret)))
```

Flags

- Executing the comparison and arithmetic instructions set flags in the CPU, which affect subsequent “conditional” instructions. There are four flags to be aware of: zero (ZF), sign (SF), carry (CF), and overflow (OF).
- **ZF** is set when the result is 0.
- **SF** is set when the most-significant bit (MSB) of the result is set.
- **CF** is set when a bit was set beyond the MSB.
- **OF** is set when one of two conditions is met:
  1. The MSB of each input is set and the MSB of the result is not set.
  2. The MSB of each input is not set and the MSB of the result is set.
- Various instructions check the flags. For example: the Jc instruction will jump if CF is set

Cmp & Je (Jz)

```
#lang racket
(require a86)
(asm-display
  (prog
    (Global 'entry)
    (Label 'entry)
    (Mov 'rax 42)
    (Cmp 'rax 42)
    (Je 'foo)
    (Mov 'rax 2)
    (Label 'foo)
    (Ret)))
```

Jne (Jnz)

```
#lang racket
(require a86)
(asm-display
  (prog
    (Global 'entry)
    (Label 'entry)
    (Mov 'rax 42)
    (Cmp 'rax 84)
    (Jne 'foo)
    (Mov 'rax 2)
    (Label 'foo)
    (Ret)))
```

Jl, Jle, Jg, Jge

```
#lang racket
(require a86)
(asm-display
  (prog
    (Global 'entry)
    (Label 'entry)
    (Mov 'rax 42)
    (Cmp 'rax 21)
    (Jl 'foo)
    (Mov 'rax 2)
    (Label 'foo)
    (Ret)))
```

Push and Pop

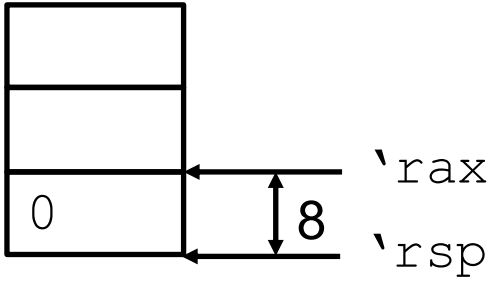
```
#lang racket
(require a86)
(asm-interp
  (prog
    (Global 'entry)
    (Label 'entry)
    (Mov 'rax 42)
    (Push 'rax)
    (Mov 'rax 0)
    ; ...
    (Pop 'rax)
    (Ret)))
```

Rsp register

- Under the hood, push and pop instructions manipulate the rsp register, which holds a pointer to the “top” of the stack
  - Push: decrement rsp, write to memory
  - Pop: read from memory, increment rsp
- If you overwrite the rsp register, you lose access to the stack (unless you stashed a pointer somewhere else)
- Need to leave stack in the state you found it in (if you push, you need to eventually pop)

Push and Pop

```
#lang racket
(require a86)
(asm-interp
   (prog
    (Global 'entry)
    (Label 'entry)
    (Mov 'rax 'rsp)
    (Push 0)
    (Sub 'rax 'rsp); 8
    (Pop 'rbx)
    (Ret)))
```



Read the stack without popping

- Mov reg (Offset rsp 0) copies first element of stack into reg
- Mov reg (Offset rsp 8) copies second element of stack into reg
- Mov reg (Offset rsp *n**8) copies (*n*+1)th element of stack into reg
- Why multiples of 8? Memory is byte-addressed.
  1 byte = 8 bits, 8 bytes = 64 bits.

Write the stack without pushing

- Mov (Offset rsp 0) reg copies reg into first element of stack
- Mov (Offset rsp 8) reg copies reg into second element of stack
- Mov (Offset rsp *n**8) reg copies reg into (*n*+1)th element of stack

Push, Pop

Push reg → Sub rsp 8
Mov (Offset rsp 0) reg

Pop reg → Mov reg (Offset rsp 0)
Add rsp 8

Label, Call, Ret

```
#lang racket
(require a86)
(asm-interp
  (prog
    (Global 'entry)
    (Label 'entry)
    (Call `f)
    (Add 'rax 1)
    (Ret)
    (Label `f)
    (Mov 'rax 41)
    (Ret)))
```

Cmov*: Conditional Move

```
Cmovz: (Cmovz 'rax 'r9)
Cmove
Cmovnz
Cmovne
Cmovl
Cmovle
Cmovg
Cmovge
Cmovo
Cmovno
Cmovc
Cmovnc
```

Cmov*

```
#lang racket
(require a86)
(asm-interp
  (prog
    (Global 'entry)
    (Label 'entry)
    (Mov 'rax 0)
    (Mov 'rcx 100)
    (Cmp 'rax -1)
    (Cmovg 'rax 'rcx)
    (Ret)))

Result:100
```

Call & ret vs Push, Pop & Jmp

```
(asm-interp
  (prog
    (Global 'entry)
    (Label 'entry)

    (Call 'f)
    ;(Lea 'rcx 'g)
    ;(Push 'rcx)
    ;(Jmp 'f)
    ;(Label 'g)
    (Add 'rax 1)
    (Ret)
```

```
(Label 'f)
    (Mov 'rax 41)
    (Ret)
    ;(Pop 'rcx)
    ;(Jmp 'rcx)
    ))
```

Example: 1+2+3+…10 = 55

```
(prog
   (Global 'entry)
   (Label 'entry)
   (Mov 'rbx 10)
   (Label 'loop)
   (Add 'rax 'rbx)
   (Cmp 'rbx 0)
   (Je 'end)
   (Sub 'rbx 1)
   (Jmp 'loop)
   (Label 'end)
   (Ret))
```

Example: 1+2+3+…10 = 55 (with function call)

```
(prog
  (Global 'entry)
  (Label 'entry)
  (Mov 'rax 10)
  (Call `f)
  (Ret)

  (Label 'f)
  (Push 'rax)
  (Sub 'rax 1)
  (Cmp 'rax 0)
  (Je 'end)
  (Call 'f)
  (Label 'end)
  (Pop 'rbx)
  (Add 'rax 'rbx)
  (Ret)))
```