CSC2/452 Computer Organization
Assembly Language

Sreepathi Pai

URCS

September 26, 2022

Outline

Outline

Assignments and Homeworks

- Assignment #1 due today
- Homework #4 out Wed
- Review class Mon Oct 3.
- Midterm next Wed, Oct 5 (in class), all material up to this Wed.

Outline

Previously

- Integers
- Floats
- Addresses
- Instructions
- The Instruction Set Architecture
  - The programmer's interface to the processor

Addresses: Summary

- Unsigned integers from 0 to $2^n - 1$ where n is size of address in bits
  - Usually $n = 64$ on modern systems
- Labels are addresses
- Addresses can be loaded into registers
  - `leaq` instruction on Intel 64 machines
- Effective addresses (the final address after any computations used to access memory) may be specified:
  - Directly (Absolute) e.g., `0x7f08e678d000` (but not in x86)
  - Indirect e.g., `mov (%rbx, %rsi, 1), 1`
  - Relative e.g., `jmp`
  - Implicit e.g., `push`

Instruction Encoding

- Instructions are encoded as multiword bitfields
  - On Intel 64, they can occupy more than 64 bits
  - Instruction encodings vary by processor
- They convey to the processor:
  - What operation to perform
  - What the operands (i.e. inputs and outputs) to that operation are
  - Operands can be registers, memory or constants
- In the Intel ISA, not all combinations of operands are valid
  - It is not fully orthogonal

Outline

Nomenclature

- Intel Processors have traditionally been known as x86
  - 8086 (their first 16-bit processor)
  - 80186, 80286
  - 80386, 80486 (their 32-bit processors)
- 80586 became the Pentium, and Intel dropped the numbering scheme
  - also 32-bit
  - Courts said you couldn't trademark numbers
- The ISA for this was usually called 'x86' by everybody or IA-32 (by Intel)

Going to 64-bits

- The first 64-bit version of the x86 was made by AMD
  - Was a new ISA based on x86 (much nicer!)
  - Therefore sometimes called ‘amd64’
  - Also referred to as ‘x86-64’
  - Intel calls their version (which is not exactly the same) ‘Intel 64‘
  - Sometimes you will see ’x64’ to refer to this architecture
- Intel’s original proposal for a 64-bit processor was called Itanium
  - ‘IA-64’, but the ISA was not widely adopted (sold about 250K each year, EOLed 2021)
  - Don’t confuse IA-32 and IA-64 – they’re not related at all!

An overview of the Intel Manuals

- Volume 1: Basic Architecture
  - Overview of all the data types, instructions, etc. that a programmer needs to know (500 pages)
  - Recommend reading this
- Volume 2: Instruction Set Reference
  - Describes every instruction, its operands, its encoding, and semantics (2356 pages)
  - Look this up when you have to
- Volume 3: System Programming Guide
  - If you're writing an OS or compiler or linker or assembler (1700 pages)
- Volume 4: Model-specific registers
  - MSRs allow you to control processors (500 pages)
  - Put them to sleep, set their operating mode, etc.
- You could order paper copies from Intel for free in the past
  - You still can, though not for free

Recall: Intel Instruction Format

B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)

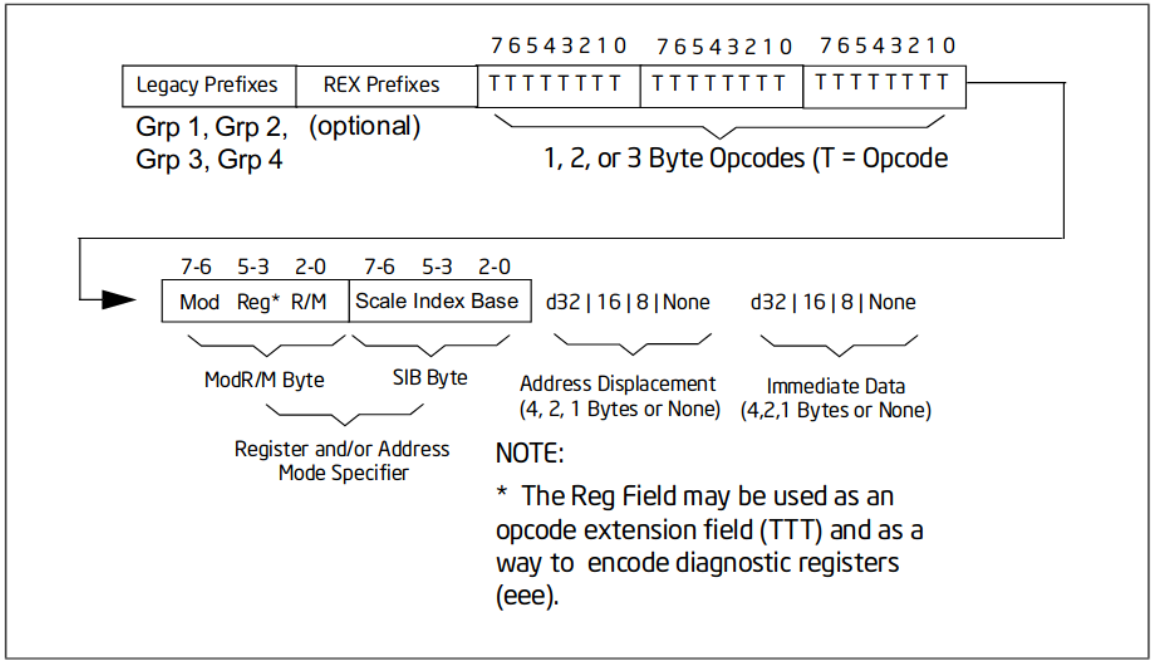


Figure B-1. General Machine Instruction Format

Source: Intel 64 and IA-32 Architectures: Software Developers Manual, Volume 2, Instruction Set Reference (A–Z), pg. 2095

The pushq instruction

```
11 0000 55                              pushq   %rbp
```

- Line 11, address 0000 of `main`
- `pushq %rbp` is encoded as `0x55`

The PUSH instruction

- `%rbp` is a 64-bit register
- That has Opcode 50+*rd*
  - That's `0x50`, and the `0` format
- Page 4-521 (i.e., 1235) in Vol 2

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	M	Valid	Valid	Push *r/m16.*
FF /6	PUSH *r/m32*	M	N.E.	Valid	Push *r/m32.*
FF /6	PUSH *r/m64*	M	Valid	N.E.	Push *r/m64.*
50+*rw*	PUSH *r16*	O	Valid	Valid	Push *r16.*
50+*rd*	PUSH *r32*	O	N.E.	Valid	Push *r32.*
50+*rd*	PUSH *r64*	O	Valid	N.E.	Push *r64.*
6A ib	PUSH *imm8*	I	Valid	Valid	Push *imm8.*
68 iw	PUSH *imm16*	I	Valid	Valid	Push *imm16.*
68 id	PUSH *imm32*	I	Valid	Valid	Push *imm32.*
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA
O	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA
ZO	NA	NA	NA	NA

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

What's *+rd*?

- **+rb, +rw, +rd, +ro —** Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. "+ro" is applicable only in 64-bit mode. See Table 3-1 for the codes.
- **+i —** A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BL	None	3	BX	None	3	EBX	None	3	RBX	None	3
AH	Not encodable (N.E.)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A
CH	N.E.	5	BP	None	5	EBP	None	5	N/A	N/A	N/A
DH	N.E.	6	SI	None	6	ESI	None	6	N/A	N/A	N/A
BH	N.E.	7	DI	None	7	EDI	None	7	N/A	N/A	N/A
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5

3-2 Vol. 2A

- *+rd* indicates EBP is 5
  - But EBP is 32-bits!
- Page 3-2 (i.e. 106) of Vol 2, see Table 3-1

In 64-bit Mode ...

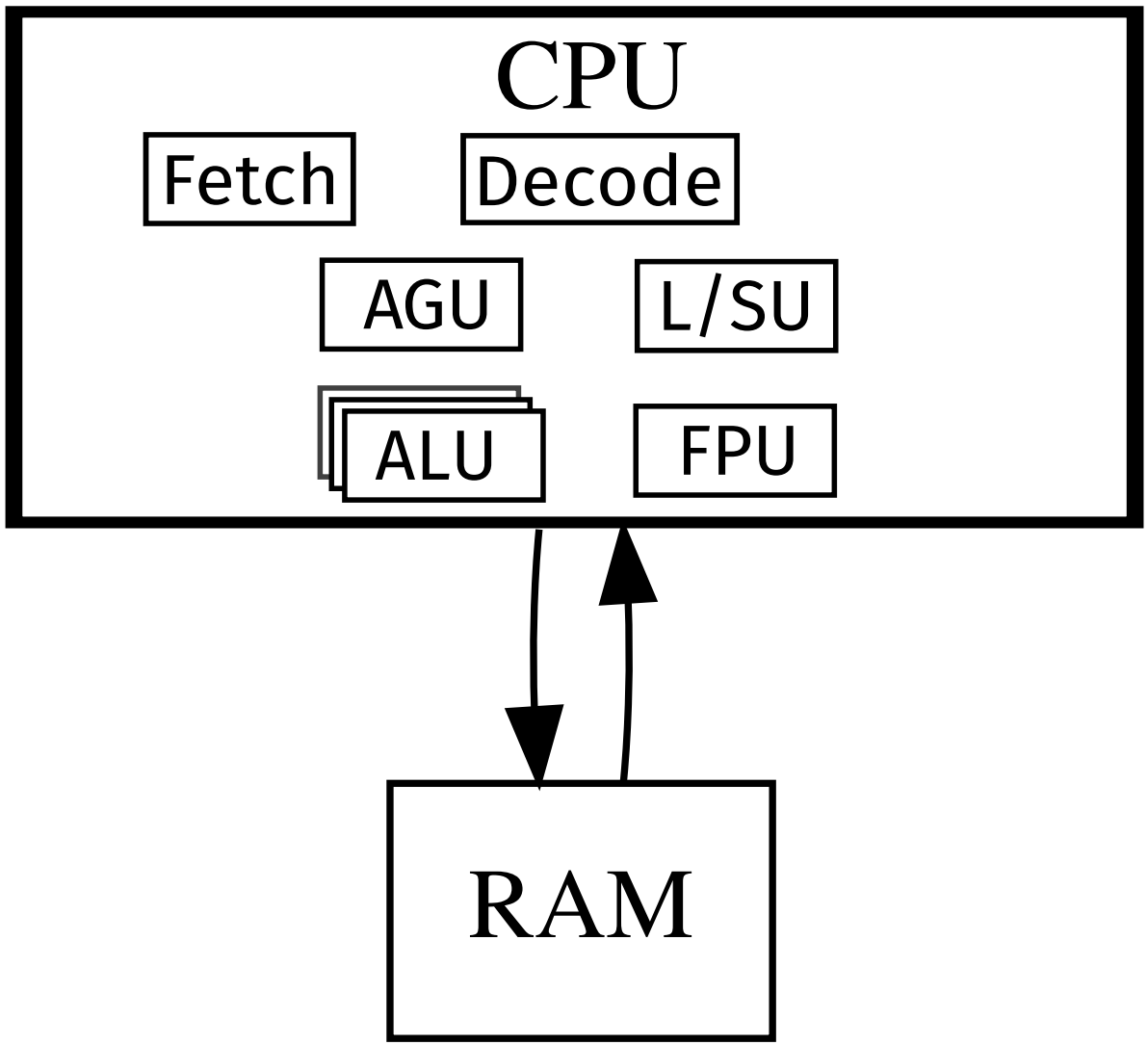
2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

- Since `PUSH` implicitly references register RSP, this makes `0x55` reference `%rbp`, not `%ebp`
  - In 64-bit mode, you can't push EBP.
- The REX prefix of `0x48` tells the processor to use 64-bit registers
  - `12 0001 4889E5 movq %rsp, %rbp`
  - It is not needed for PUSH
- Page 2-B (i.e. 44) of Vol 2 (REX prefixes)

CPU: The Decode Unit



Outline

Features of High-level Languages (HLL)

- Variables
- Arrays
- Complex Expressions (large number of operands)
  - Arithmetic Operations, Logical Operations, etc.
- Block structure - { and } in C-like languages
- Conditionals `if-then-else`
- Loops `while`, `for`
- Functions

Features of Assembly Languages

- Memory
- Registers
- Expressions are very simple, maybe up to 3 operands
  - Need to break up complex expressions into simple expressions
- No block structure
- No direct equivalent for conditionals
- No direct equivalent for loops
- Limited support for functions

Translating Variables

- Typically variables in programs like C end up in either:
  - Memory
  - Registers
- Memory can be seen as two logical regions – heap, and stack
  - Heap usually stores global variables and data
  - Stack usually stores (function) local variables
- In assembly language, a variable is in:
  - a register, if instruction uses a register operand, e.g. `%rax` (or any other register),
  - heap, if instruction uses an indirect memory operand, e.g. `(%rbx)`
  - stack, if an instruction uses an indirect memory operand relative to `%rbp` or `%rsp`, e.g. `-4(%rbp)`
- Accessing memory is slow, so a variable may be loaded from memory into a register before operating on it
  - In which case, for some time, it exists in both memory and registers

Example

```
int sum(int a, int b) {
  int c = 0;

  c = a + b;

  return c;
}
```

```
0000000000000000 <sum>:
   0:   55                          push   %rbp
   1:   48 89 e5                    mov    %rsp,%rbp
   4:   89 7d ec                    mov    %edi,-0x14(%rbp)  # A r->s
   7:   89 75 e8                    mov    %esi,-0x18(%rbp)  # B r->s
   a:   c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)   # C s
  11:   8b 55 ec                    mov    -0x14(%rbp),%edx  # A s->r
  14:   8b 45 e8                    mov    -0x18(%rbp),%eax  # B s->r
  17:   01 d0                       add    %edx,%eax         # A + B
  19:   89 45 fc                    mov    %eax,-0x4(%rbp)   # store into C
  1c:   8b 45 fc                    mov    -0x4(%rbp),%eax   # C s->r (rval)
  1f:   5d                          pop    %rbp
  20:   c3                          retq
```

- `r->s`, register to stack. `s->r`, stack to register
- `rval`, return value (must be stored in %eax)

Translating Expressions

- Most instructions accept only 1, 2 or 3 operands
- Example: ADD instruction takes two operands
  - `ADD %r1, %r2`
  - `%r2 = %r1 + %r2`
- An expression like `s = a + b + c` needs to be broken down to fit this instruction
  - Do this by introducing temporaries

```
tmp = a + b;
tmp = tmp + c;
s = tmp;
```

Translating Expressions: Example

```
0000000000000000 <sum3>:
int sum3(int a, int b, int c) {
   0:    55                          push   %rbp
   1:    48 89 e5                    mov    %rsp,%rbp
   4:    89 7d ec                    mov    %edi,-0x14(%rbp) # A
   7:    89 75 e8                    mov    %esi,-0x18(%rbp) # B
   a:    89 55 e4                    mov    %edx,-0x1c(%rbp) # C
  int s = 0;
   d:    c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)  # S

  s = (a + b + c);
  14:    8b 55 ec                    mov    -0x14(%rbp),%edx
  17:    8b 45 e8                    mov    -0x18(%rbp),%eax
  1a:    01 c2                       add    %eax,%edx        # %edx = A + B
  1c:    8b 45 e4                    mov    -0x1c(%rbp),%eax
  1f:    01 d0                       add    %edx,%eax        # C + %edx
  21:    89 45 fc                    mov    %eax,-0x4(%rbp)  # store into S

  return s;
  24:    8b 45 fc                    mov    -0x4(%rbp),%eax
}
  27:    5d                          pop    %rbp
  28:    c3                          retq
```

Translating Conditionals

```
int max(int a, int b) {

  if(a > b) {
    return a;
  } else {
    return b;
  }

}
```

Assembly Language Conditionals: Three components

- Comparison functions
- Conditional jumps
- Unconditional jumps

Disassembly of max

```
0000000000000000 <max>:
   0:    55                       push   %rbp
   1:    48 89 e5                 mov    %rsp,%rbp
   4:    89 7d fc                 mov    %edi,-0x4(%rbp)
   7:    89 75 f8                 mov    %esi,-0x8(%rbp)
   a:    8b 45 fc                 mov    -0x4(%rbp),%eax
   d:    3b 45 f8                 cmp    -0x8(%rbp),%eax   # COMPARISON
  10:    7e 05                    jle    17 <max+0x17>     # COND. JUMP
  12:    8b 45 fc                 mov    -0x4(%rbp),%eax
  15:    eb 03                    jmp    1a <max+0x1a>     # UNCOND. JUMP
  17:    8b 45 f8                 mov    -0x8(%rbp),%eax
  1a:    5d                       pop    %rbp
  1b:    c3                       retq
```

Comparison

```
 a:    8b 45 fc                mov    -0x4(%rbp),%eax
 d:    3b 45 f8                cmp    -0x8(%rbp),%eax   # COMPARISON
10:    7e 05                   jle    17 <max+0x17>     # COND. JUMP
...
17:    8b 45 f8                mov    -0x8(%rbp),%eax   # RETURN B
```

- We know `a` is on the stack at `-0x4(%rbp)`
- We know `b` is on the stack at `-0x8(%rbp)`
- The `cmp` instruction compares `b` with `%eax` (which contains `a`)
  - For `cmp x, y`, the `cmp` instruction calculates `y - x`
- The results of the `cmp` operation are stored in the EFLAGS register. Of relevance to `jle`:
  - ZF: Zero flag: set to 1 if `y - x == 0`
  - OF: Overflow flag: set to 1 if `y - x` underflowed or overflowed
  - SF: Sign flag: set to sign bit of `y - x`

jle

- Part of the family of `j`cc instructions
  - cc is conditional code
- Jump If Less Or Equal
- If ZF is 0, then $y = x$.
- If OF $\neq$ SF then, $y < x$
  - To understand this, work out all cases of $y < x$ where they are +/+,-/-,+/-,-/+
  - Note that at machine level, two's complement integers "wrap-around" on overflow and underflow
- Here, the conditional jump to `max+0x17` occurs if $a <= b$
  - Otherwise control "falls through" to the next instruction

Conditionals: Full translation

- Evaluate the condition and jump to the Else part
  - Or fall through to the Then part

```
 a:    8b 45 fc                 mov    -0x4(%rbp),%eax
 d:    3b 45 f8                 cmp    -0x8(%rbp),%eax   # COMPARISON
10:    7e 05                    jle    17 <max+0x17>     # COND. JUMP
```

- The Then Part

```
12:    8b 45 fc                 mov    -0x4(%rbp),%eax
15:    eb 03                    jmp    1a <max+0x1a>     # UNCOND. JUMP
```

- The Else Part

```
17:    8b 45 f8                 mov    -0x8(%rbp),%eax
```

- Code immediately after Else

```
1a:    5d                       pop    %rbp
```

Loops

```
int div(int a, int b) {
  int q = 0;

  while(a - b > 0) {
    a = a - b;
    q = q + 1;
  }

  return q;
}
```

(Ignore what this function is trying to do)

Loops: Removing Structure

```
int div2(int a, int b) {
  int q = 0;

  goto loop_test;

loop_body:
  a = a - b;
  q = q + 1;

loop_test:
  if((a - b > 0))
    goto loop_body;

loop_exit:  /* not required, for clarity only */

  return q;
}
```

- We can convert a `while` loop into *unstructured* form using `goto` and `if`

Unstructured Loop Translation

- `goto` is an unconditional `jmp`
- The `if(cond) goto` form is just a conditional jump

Translation

```
000000000000002c <div2>:
  ...
  3d:   eb 0b             jmp    4a <div2+0x1e>       # goto loop_test
  3f:   90                nop                         # loop_body:
  40:   8b 45 e8          mov    -0x18(%rbp),%eax
  43:   29 45 ec          sub    %eax,-0x14(%rbp)
  46:   83 45 fc 01       addl   $0x1,-0x4(%rbp)
  4a:   8b 45 ec          mov    -0x14(%rbp),%eax     # loop_test: a - b
  4d:   2b 45 e8          sub    -0x18(%rbp),%eax
  50:   85 c0             test   %eax,%eax            # is a - b > 0?
  52:   7f eb             jg     3f <div2+0x13>       # jump if greater
  54:   90                nop
  55:   8b 45 fc          mov    -0x4(%rbp),%eax
  58:   5d                pop    %rbp
  59:   c3                retq
```

- `test` performs a logical and of its two operands and sets ZF and SF
  - `sub` sets the OF flag
- `jg` jumps if ZF=0 and SF=OF

Translating `for` loops

```
int iter(int a, int b) {
  int i;

  for(i = 0; i < 10; i++) {
    a = a + b;
  }

  return i;

}
```

De-structuring `for` loops

```
int iter2(int a, int b) {
  int i;

loop_head:
  i = 0;
  goto loop_test;

loop_body:
  a = a + b;

  i++;   /* loop update */

loop_test:
  if(i < 10)
    goto loop_body;

loop_exit:

  return i;
}
```

Translating `for` loops

```
0000000000000028 <iter2>:
  ...
  32:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)   # i = 0
  39:   eb 0b                   jmp    46 <iter2+0x1e>   # goto loop_test

  3b:   90                      nop                      # loop_body:
  3c:   8b 45 e8                mov    -0x18(%rbp),%eax
  3f:   01 45 ec                add    %eax,-0x14(%rbp)  # a = a + b
  42:   83 45 fc 01             addl   $0x1,-0x4(%rbp)   # i++

  46:   83 7d fc 09             cmpl   $0x9,-0x4(%rbp)   # loop_test:
  4a:   7e ef                   jle    3b <iter2+0x13>   # if(i < 10) goto loop_body

  4c:   90                      nop
  ...
```

Basics of translating HLLs to Assembly (so far)

- Simplify expressions
- Find locations for variables
- Destructure loops
  - Use conditional and unconditional jumps

Handling Function Calls

- How to pass arguments to function?
- How to jump to a function?
- How to come back to just after call location?
  - How does `ret` know where to return to?
- How to receive the return value from a function?

References

- Read Chapter 3 of the textbook
  - Esp. the Figure detailing all the registers