CS2011 Notes

Introduction to Machine Organization and Assembly Language

Karthik Seetharaman

WPI B Term 2021

Contents

1 Preface

These notes were taken during my attendance of the CS2011 course as a dual enrolled student at WPI. Dates in the table of contents refer to the date of the lecture those notes correspond to. The material in these notes has not been checked for completeness or correctness. The notes also contain materials (screenshots) directly from the class slides.

The course is based on the course 15-213 at CMU. The official textbook for the course was the third edition of *Computer Systems: A Programmer's Perspective* by Randal Bryant and David O'Halloran. The official course description of the course is below.

Course Description: This course introduces students to the structure and behavior of modern digital computers and the way they execute programs. Machine organization topics include the Von Neumann model of execution, functional organization of computer hardware, the memory hierarchy, caching performance, and pipelining. Assembly language topics include representations of numbers in computers, basic instruction sets, addressing modes, stacks and procedures, low-level I/O, and the functions of compilers, assemblers, linkers, and loaders. The course also presents how code and data structures of higher-level languages are mapped into the assembly language and machine representations of a modern processor. Programming projects will be carried out in the C language and the assembly language of a modern processor. Recommended background: CS 2301 or CS 2303 (Systems Programming Concepts), or a significant knowledge of C/C++.

2 October 25, 2021

§2.1 Goals of Course

1. Understand the X/86-64 hardware; how does it:
   a) Store and represent data
   b) Execute instructions
   c) Organize physical memory
2. Understand the Linux operating system; how does it:
   a) Manage memory
3. Understand linker/loader; how does it:
   a) Manage modules to be executed
4. Understand the C compiler; how does it:
   a) Translate C
   b) Generate assembly and machine language

3 October 26, 2021

§3.1 Representing Information As Bits

Information is represented in the form of bits (0 or 1). Electronic devices represent bits using different voltage levels in a circuit.

Definition 3.1.1. A **byte** is 8 bits. It is represented in binary, so it takes values from 00000000_2 to 11111111_2, inclusive.

> **Remark 3.1.2.** A byte takes values from 0 to 255 in decimal and 00 to FF in hexadecimal, inclusive.

Roughly, memory in a device is organized by bytes (a very large array of bytes).

Definition 3.1.3. When a program is being executed, the system provides space private to that program. This is called an **address space** (or virtual address space). This allocation is done by the compiler and runtime system.

Definition 3.1.4. The size of a pointer on a machine is its **word size**. This determines the maximum size of the address space.

> **Remark 3.1.5.** Older machines use 32 bit words, but these can be too small for some memory-intensive applications. Many systems, including X86-64, now use 64 bits as their word size.

Question 3.1.6. How do we order bytes within a multi-byte word in memory?

We can use **Big Endian**, as in the internet, where the least significant byte has the highest address. X86 uses **Little Endian**, where the least significant byte has the lowest address.

Assembly code is converted into a list of bytes for the machine to look at and execute. Numbers in the assembly code are padded to 32 bits, split into bytes, and reversed (to store in Little Endian) to get the corresponding list in the machine instructions.

4 October 28, 2021

§4.1 Boolean Algebra

We encode "True" as 1 and "False" as 0. In Boolean algebra, all variables are either true or false. There are 4 basic operations in Boolean algebra, which are applied bitwise (when applying to numbers with more than one bit, each pair of corresponding bits has the operation applied to it, padding with zeroes when necessary):

1. "And": $A\&B = 1 \iff A = B = 1$
2. "Or": $A|B = 1 \iff (A, B) \neq (0, 0)$
3. "Not": $\tilde{}A = 1 \iff A = 0$
4. "Exclusive-or (XOR)": $A\hat{}B \iff (A, B) = (1, 0), (0, 1)$

Boolean algebra can be applied to digital systems: a closed switch can be encoded as 1 and an open switch as 0.

A width w vector of bits, represents subsets of $\{0, 1, \ldots, w-1\}$.

Example 4.1.1

Let $w = 8$. Then, $\{0, 3, 5, 6\}$ is represented as 01101001 and $\{0, 2, 4, 6\}$ is represented as 01010101.

Then, we can intersect two subsets of $\{0, 1, \ldots, w-1\}$ by taking the "and" of their bit vectors. Similarly, we can union by taking their "or" and their symmetric difference by taking their "XOR". Using "not" on a set's bit vector gives its complement.

Remark 4.1.2. In C, these four operations are available to use on any integral type (long, int, short, char, etc.). Arguments are viewed as bit-vectors and operations are applied bitwise.

Example 4.1.3

$\tilde{}$0x41 = 0xBE, $\tilde{}$0x00=0xFF.

Example 4.1.4

0x69 & 0x55 = 0x41.

Example 4.1.5

0x69 | 0x55 = 0x7D.

§4.1.1 Logical Operations

Definition 4.1.6. C has **logical operations** &&, ||, and !, which correspond to and, or, and not respectively. These operators view 0 as False, anything nonzero as True, and always return 0 or 1.

Example 4.1.7

$0x41 = 0x00, !0x00 = 0x01, !!0x41 = 0x01$.

Example 4.1.8

0x69 & & 0x55 = 0x01.

Example 4.1.9

0x69 || 0x55 = 0x01.

§4.1.2 Shift Operations

Definition 4.1.10. $x << y$ means to shift the bit-vector x left y positions, throwing away extra bits on the left and padding with zeroes on the right. This is a **left shift**.

Definition 4.1.11. Similarly, $x >> y$ is a **right shift** and means to shift the bit-vector x right y positions, padding with zeroes on the left.

Definition 4.1.12. The above two definitions are **logical shifts**. There are also **arithmetic shifts**, where instead of padding with zeroes, the most significant bit is replicated in the now empty slots.

Example 4.1.13

Let $x = 10100010$. Then, $x >> 2$, logically, is 00101000, and arithmetically, it is 11101000.

Remark 4.1.14. Note that the directions left and right for shifting are independent of Little Endian and Big Endian. Shifting applies to an integral data item as a whole, regardless of how it is stored in memory.

§4.2 Integers

Definition 4.2.1. Let $X = (x_0, x_1, \ldots, x_{w-1})$ be a bit-vector of length w. Then, the unsigned integer value of this bit-vector is

$$\sum_{i=0}^{w-1} x_i \cdot 2^i.$$

Definition 4.2.2. Let $X = (x_0, x_1, \ldots, x_{w-1})$ be a bit-vector of length w. Then, the **two's complement** representation of X is the signed integer value of X. Specifically, this is

$$-x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i.$$

We call x_{w-1} the **sign bit**, since the most significant bit indicates the sign.

The maximum value of an unsigned n bit integer is $2^n - 1$. For signed integers, the maximum value is $2^{n-1} - 1$ and the minimum value is -2^{n-1}.

5 October 29, 2021

§5.1 Conversion and Casting

To convert between signed and unsigned integers, the bit-vector stays the same and the numbers are evaluated differently (two's complement vs unsigned evaluation).

Theorem 5.1.1

Let X be a bit-vector of length n, and let x be its signed value and ux its unsigned value. Then, $ux = x$ if $x \geq 0$ and $ux = x + 2^n$ if $x < 0$.

§5.1.1 Casting in C

Constants, by default, are signed. If they are unsigned, they will have U as a suffix (e.g. 0U).

If an expression has unsigned and signed values, signed values implicitly cast to unsigned. This includes comparison operations.

Example 5.1.2

Examples of comparisons:

1. 0 == 0U (unsigned evaluation)
2. -1 < 0 (signed evaluation)
3. -1 > 0U (unsigned evaluation)
4. 2147483647 > -2147483647-1 (signed evaluation)
5. 2147483647U < -2147483647-1 (unsigned evaluation)
6. -1 > -2 (signed evaluation)
7. (unsigned)-1 > -2 (unsigned evaluation)
8. 2147483647 < 2147483648U (unsigned evaluation)
9. 2147483647 > (int) 2147483648U (signed evaluation)

Remark 5.1.3. When we typecast in an expression (i.e. (unsigned) or (int) before an integer), we only change the value of that integer.

6 November 1, 2021

§6.1 Expanding and Truncating Integers

Definition 6.1.1. Say we have some w-bit signed integer x and we want to convert it to a $w+k$-bit integer with the same value. Then, we make k copies of the sign bit and append them to the left of x. This converts from a smaller to larger integer data type while keeping the signed value of the bit-vector constant. This is **expansion**.

Definition 6.1.2. To **truncate** the last k bits of a number, we essentially take x modulo 2^k.

> **Remark 6.1.3.** For signed numbers, the truncation may have a different sign than the original!

§6.1.1 Complementing and Incrementing Integers

Claim 6.1.4 — In the two's complement representation of an integer x, we have

$$\tilde{x} + 1 == -x.$$

Of course, this means

$$\tilde{x} + x == -1.$$

§6.2 Integer Operations

§6.2.1 Addition

Definition 6.2.1 (Addition)**.** Say we are adding two unsigned operands with w bits, but the true sum of the integers has $w+1$ bits. In this case, the machine just gets rid of the highest-order bit. This is equivalent to truncating the true sum to the last w bits. Two's complement addition has identical bit-level behavior, except the resulting truncated sum is evaluated as a signed integer.

> **Remark 6.2.2.** Integers (signed or unsigned) under the modular addition that computers perform forms an abelian group ($\mathbb{Z}/2^w\mathbb{Z}$ for some given word size).

§6.2.2 Multiplication

Definition 6.2.3 (Multiplication)**.** Multiplication is a combination of shifting and adding. Each bit is either a 0 or 1, so just like how we multiply numbers normally, each digit of the second number is multiplied by the first number, the appropriate shifts are done, then everything is added together. The first step only requires replication of the first numbers since all bits are 0 or 1. Consider the example show in Figure 6.1.

Decimal	Hex	Binary
15213	3B 6D	00000000 00000000 00111011 01101101
2011	07 DB	00000111 11011011
		00111011 01101101
		0 01110110 1101101
		001 11011011 01101
		0011 10110110 1101
		001110 11011011 01
		0011101 10110110 1
		00111011 01101101
		0 01110110 1101101
		00 11101101 101101
30593343	1 D2 D1 3F	0001 11010010 11010001 00111111

Figure 6.1: An example of multiplying 15213 by 2011.

Remark 6.2.4. In modern machines, multiplication takes 10 or more machine cycles. However, this is easily pipelined (split into several parallel processes).

For small constant multipliers, compilers have optimizations. Note that $u << k$ is the same as $u \cdot 2^k$, for both signed and unsigned integers. Shifting and adding is fast, so the C compiler automatically converts constant multipliers into shift-add code. For example,

$$12u = ((u) + (u << 1)) << 2.$$

7 November 4, 2021

§7.1 Integer Operations (Continued)

§7.1.1 Division

Definition 7.1.1 (Division)**.** This is a similar principle to multiplication, where the divisor is subtracted from the higher-order bits of the dividend, it is shifted, and the next bit is brought down.

This is very costly in number of cycles, costing 30 or more for an integer divide. Furthermore, it's not very amenable to pipelining.

If we want to divide an unsigned integer by a power of 2, we can do $u >> k$. If we do the same for a signed integer, we round the wrong direction when using a negative integer.

Definition 7.1.2. To fix the above issue, we add the **bias** $2^k - 1$ to make the result of the division tend toward 0. This means dividing a negative integer x by 2^k is actually computed as

$$\left\lfloor \frac{x + 2^k - 1}{2^k} \right\rfloor.$$

> **Remark 7.1.3.** In C, the above formula is written as $(x + (1 << k) - 1) >> k$.

§7.2 Floats

Definition 7.2.1. Say we have some binary number $b_i b_{i-1} \dots b_2 b_1 b_0 . b_{-1} b_{-2} \dots b_{-j}$, where the point is a **binary point**. This is evaluated as the rational number

$$\sum_{k=-j}^{i} b_k \cdot 2^k.$$

> **Remark 7.2.2.** In the same way as with integers, we can divide by 2 by shifting right and multiply by 2 by shifting left.

Representing numbers in this form means we can only exactly represent rational numbers with denominator 2^k. Other rationals have repeating bit representations. Furthermore, this method does not allow for really big or really small numbers.

§7.2.1 IEEE Floating Point Standard

> **Remark 7.2.3.** This is also known as IEEE Standard 754.

Definition 7.2.4. The numerical form of a floating point number is

$$(-1)^s \cdot M \cdot 2^E.$$

Here, s is the **sign bit**, 1 if the float is negative and 0 if it is positive. The **mantissa** M is normally a fractional value in the range $[1.0, 2.0)$. The **exponent** E weights the value by a power of two.

There are many precisions of floats. The 32-bit precision keeps 1 sign bit, 8 exp bits, and 23 frac bits. The 64-bit precision keeps 1 sign bit, 11-bit exp bits and 52 frac bits.

§7.2.2 Normalized Values

Definition 7.2.5. A float is **normalized** if the exp field is not all zeroes or all ones.

In the case of a normalized float, the exponent is coded as a biased value.

Definition 7.2.6. The **bias** of a float is $2^{k-1} - 1$, where k is the number of exponent bits. In single-precision, this would be 127, and in double-precision, this would be 1023.

The exponent E of a float is coded as the value of the exp field minus the bias.
The mantissa is coded with an implied leading 1 (since it is between 1 and 2).

§7.2.3 Denormalized Values

If the exp field is all zeroes, the exponent value is 1 minus the bias. In this case, the mantissa is encoded with implied leading bit zero instead of implied leading bit one. If the frac is also zero, the whole float is zero.

> **Remark 7.2.7.** Depending on the sign bit, we have a positive zero and negative zero!

If the exp field is all ones and the frac field is zero, the float is interpreted as infinity. If the frac field is nonzero, the float is interpreted as NaN (not a number).

8 November 5, 2021

§8.1 Floating Point Operations

> **Remark 8.1.1.** Unsigned integer comparison almost works for floats, except for the special cases of NaNs. You must first compare the sign bits.

The basic idea of floating point operations is to compute an exact result and then make it fit into the desired precision.

§8.1.1 Rounding

There are many ways we can round:

1. Towards zero
2. Round down
3. Round up
4. Nearest even (default)

Definition 8.1.2. **Nearest even rounding** rounds to the closest integer except for when the float is halfway between two integers. In this case, the float rounds to the nearest even integer. To apply this to rounding in other decimal places, simply round so that the least significant digit is even.

> **Remark 8.1.3.** All other modes of rounding mentioned in the list are statistically biased; the sum of a set of positive numbers, rounded, will always be over or under-estimated under the other modes.

In binary, "even" is when the least significant bit you are rounding towards is 0. "Halfway" is when the bits to the right of the rounding position are 1 followed by a bunch of zeroes.

§8.1.2 Creating Floats

There are three steps to creating a float from an unsigned number:

1. Set the binary point so that all numbers are of the form 1.xxx. If necessary, decrement the exponent by shifting left. This is **normalization**.
2. There are now too many fractional bits, so we must round.
3. If the rounding caused overflow, shift right once and increment the exponent. This is **postnormalization**.

To round, we go through the following algorithm:

1. Designate the least significant non-fractional bit as the **guard bit**, the first bit removed as the **round bit**, and the **sticky bit** as the OR of the remaining bits.
2. We round up if the round and sticky bits are both 1, and we round to even if the guard bit and round bits are 1 and the sticky bit is 0. Otherwise, we just truncate.

§8.1.3 Multiplication

Say we have two floating point numbers $F_1 = (-1)^{S_1} \cdot M_1 \cdot 2^{E_1}$ and $F_2 = (-1)^{S_2} \cdot M_2 \cdot 2^{E_2}$. Then,

$$F_1 \cdot F_2 = (-1)^{S_1 \hat{} S_2} \cdot (M_1 \cdot M_2) \cdot 2^{E_1 + E_2}.$$

If $M \geq 2$, we shift M right and increment E. We also have to round M to fit the fractional bit precision. In general, the biggest chore is multiplying M_1 and M_2.

§8.1.4 Addition

Say we have two floating point numbers $F_1 = (-1)^{S_1} \cdot M_1 \cdot 2^{E_1}$ and $F_2 = (-1)^{S_2} \cdot M_2 \cdot 2^{E_2}$. Then, the exponent of the result is the greater of the 2 and the sign and significand are found by doing a signed align and add. Again, we deal with overflow in a similar way to multiplication.

9 November 8, 2021

§9.1 Machine Level Programming - Basics

Definition 9.1.1. **Architecture** is the design of a computer.

Definition 9.1.2. **Instruction Set Architecture** (ISA) refers to the parts of a processor design which understand and write assembly code. This includes actions and registers.

Example 9.1.3

Examples of ISAs include Intel x86, IA32, Itanium, and x86-64.

Definition 9.1.4. **Microarchitecture** is the implementation of the architecture, such as cache sizes and core frequency.

Definition 9.1.5. **Machine code** is the byte-level programs which a processor executes.

Definition 9.1.6. **Assembly code** is textual representation of machine code.

§9.1.1 Assembly/Machine Code View

Figure 9.1 shows the assembly/machine code view.

The processor contains the following three things:

Definition 9.1.7. The **Program Counter**, or PC, contains the address of the next instruction. The x86-64 calls this the RIP (register instruction pointer).

Definition 9.1.8. The **register file** consists of heavily used program data. It consists of 16 **registers**, each of which have 64 bits of memory that are very fast to extract data from.

Definition 9.1.9. **Condition codes** store statuses of the most recent arithmetic or logical operations. These are 1-bit registers and are useful for conditionals.

Definition 9.1.10. The computer's **memory** is a byte-addressable array. It contains a stack to support functions.

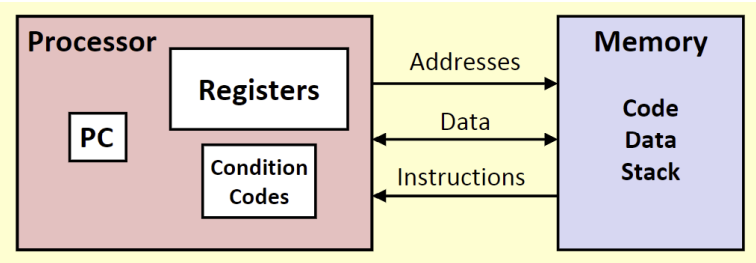

Figure 9.1: A view of the processor and memory.

§9.1.2 Von Neumann Cycle

The Von Neumann Cycle is the way computers generally perform programs.

Definition 9.1.11. The **Von Neumann Cycle** is as follows:

1. Fetch an instruction from memory and increment the program counter.
2. Decode the instruction and read the appropriate registers.
3. Perform one integer operation.
4. Access memory.
5. Write to the register.
6. Repeat.

§9.1.3 Assembly Data Types and Operations

Assembly code supports integer data of 1, 2, 4, or 8 bytes as well as floating point data of 4, 8, or 10 bytes.

Definition 9.1.12. **Code** is just byte sequences that encode instructions for the computer to execute!

Operations can move and copy data between the memory and registers. There are also operations that perform arithmetic or logical functions, as well as operations that transfer control of the program (conditionals and such).

§9.1.4 Compiling Into Assembly

A given .c file compiles into a .s file when compiled. The .s file contains the generated x86-64 assembly code from the .c file.

Definition 9.1.13. This .s file is then fed into an **assembler**, which translates the code into an **object file**, with the suffix .o. This contains a binary encoding of each instruction the computer needs to execute.

Definition 9.1.14. The **linker** resolves references between files. This includes references to **static run-time libraries**, which include statements like printf (built on compilation), as well as **dynamically linked libraries** (accessed when needed).

Example 9.1.15

Say we have the C code

```
*dest = t;
```

This translates into the assembly code

```
movq %rax, (%rbx)
```

which translates into the object code

```
0x40059e:  48 89 03
```

The object code is a three-byte instruction stored at the given address (0x40059e).

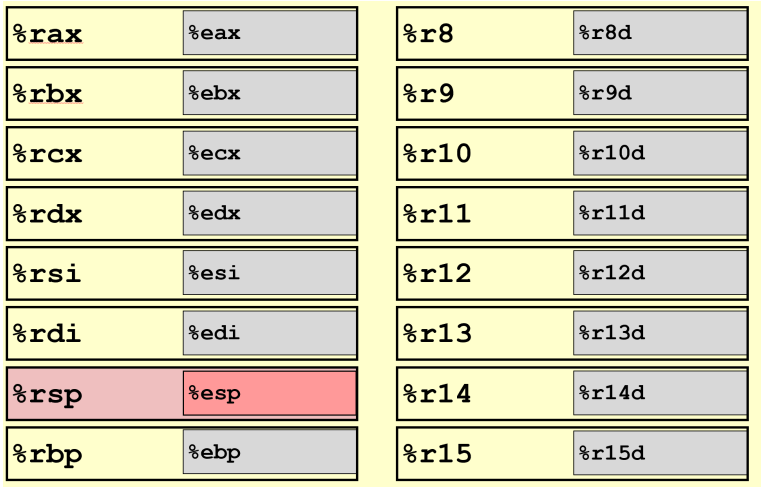

Figure 9.2: The 16 x86-64 registers. On 32-bit machines, the names are different; these are in the gray boxes.

Definition 9.1.16. The **disassembler** takes object code and outputs an approximation of the assembly code. This is accessed via the "objdump" command.

> **Remark 9.1.17.** Disassembly can also be performed within the gdb debugger, with the "disassemble" command.

§9.2 Registers

There are 16 registers in x86-64, shown in Figure 9.2. These are all 64-bit registers.

Definition 9.2.1. To move data, we use the "movq" instruction in the format "movq (source) (destination)". These operands can be constant integer data to constant integer data (which are prefixed with $), from register to register, or from memory to register (or vice-versa).

Example 9.2.2

Here are examples of movq:

```
1 movq $0x400, $-533
2 movq %rax, %r13
3 movq (%rax)
```

10 November 9, 2021

§10.1 Assembly Basics

§10.1.1 More Registers

Putting parentheses around a register name is like dereferencing it. This is like pointers in C!

Sometimes we see D(R), where R is a register. This constant displacement D represents an offset, so the start of the memory region we want is not at the start at the register.

Definition 10.1.1. The most complete form of memory addressing is $D(R_b, R_i, S)$, which refers to the memory address at

$$R_b + S \cdot R_i + D.$$

R_b is the **base register**, R_i is the **index register**, S is the **scale**, and D is the **displacement**.

Remark 10.1.2. S is always 1, 2, 4, or 8.

Example 10.1.3

Say the %rdx register is at address 0xf000 and the %rcx register is at address 0x0100. Then, we can compute the following examples:

1. 0x8(%rdx) = 0xf000 + 0x8 = 0xf008
2. (%rdx, %rcx) = 0xf000 + 0x100 = 0xf100
3. (%rdx, %rcx, 4) = 0xf000 + 4 · 0x100 = 0xf400
4. 0x80(,%rdx, 2) = 2 · 0xf000 + 0x80 = 0x1e080

Remark 10.1.4. The first argument of a function is always passed into the %rdi register, while the second argument is always passed into the %rsi register.

§10.1.2 Arithmetic and Logical Operations

Definition 10.1.5. The "leaq" operation (load effective address), used as "leaq (source) (destination)," computes an address without a memory reference.

Remark 10.1.6. The operation computes arithmetic expressions of the form $x + ky$ for $k = 1, 2, 4, 8$.

Example 10.1.7

Say we want to multiply x by 12. Then, the compiler generates:

```
leaq (%rdi, $rdi, 2), $rax
salq $2, %rax
```

The salq operation shifts the contents of %rax to the left 2 bits.

Definition 10.1.8. Here are some additional arithmetic operations with two operands:

1. addq (src) (dest) computes dest = dest + src
2. subq (src) (dest) computes dest = dest - src
3. imulq (src) (dest) computes dest = dest * src
4. salq (src) (dest) computes dest = dest << src
5. sarq (src) (dest) computes dest = dest >> src (arithmetic shift)
6. shrq (src) (dest) computes dest = dest >> src (logical shift)
7. xorq (src) (dest) computes dest = dest ^ src
8. andq (src) (dest) computes dest = dest & src
9. orq (src) (dest) computes dest = dest | src

Remark 10.1.9. These operations do not make distinctions between signed and unsigned ints.

Definition 10.1.10. Here are some arithmetic operations with one operand:

1. incq (dest) computes dest = dest + 1
2. decq (dest) computes dest = dest - 1
3. negq (dest) computest dest = -dest
4. notq (dest) computes dest = ~dest

Definition 10.1.11. The "b" suffix at the end of an instruction applies to byte quantities. The "w" suffix is for words, "l" for longs, and "q" for quad-words. These are for 8-bit, 16-bit, 32-bit, and 64-bit operands respectively (or char, short, int, long int in C).

11 November 11, 2021

§11.1 Condition Codes

Definition 11.1.1. Some registers, like %rax, contain temporary program data. The runtime stack is located in the %rsp register. The pointer to the next instruction (the **instruction pointer**) is located in the %rip register. The four **condition codes** contain information about the previous instruction executed.

Definition 11.1.2. The **CF** condition code is the carry flag (for unsigned operations). This is basically unsigned overflow.

Definition 11.1.3. The **SF** condition code is the sign flag. This is set if the sign bit is 1.

Definition 11.1.4. The **ZF** condition code is the zero flag. This is set for zero values.

Definition 11.1.5. The **OF** condition code is the overflow flag (for signed operations).

These condition codes are set as an implicit side effect of most arithmetic or logical operations.

They can also be explicitly set through operations such as "cmpq" (comparing two numbers without saving the difference) or "testq" (compute the and of two ints without setting destination). These operations will just toggle the condition codes.

§11.1.1 Reading Condition Codes

Definition 11.1.6. The **SetX Instructions** set low-order byte sof the destination to 0 or 1 based on the combinations of condition codes. The remaining 7 bytes are not altered.

1. sete has condition ZF (equal/zero)
2. setne has condition ~ZF (not equal/not zero)
3. sets has condition SF (negative)
4. setns has condition ~SF (nonnegative)
5. setg has condition ~(SF ^ OF) & ~ZF (greater than)
6. setge has condition ~(SF ^ OF) (greater than or equal to)
7. setl has condition SF ^ OF (less than)
8. setle has condition has condition (SF ^ OF) — ZF (less than or equal to)
9. seta has condition ~CF & ~ZF (unsigned above)
10. setb has condition CF (unsigned below)

§11.2 Conditional Branches

Definition 11.2.1. The **jX Instructions** jump to different parts of the code depending on condition codes.

1. jmp has condition 1
2. je has condition ZF (equal/zero)
3. jne has condition ~ZF (not equal/not zero)
4. js has condition SF (negative)
5. jns has condition ~SF (nonnegative)
6. jg has condition ~(SF ^ OF) & ~ZF (greater than)
7. jge has condition ~(SF ^ OF) (greater than or equal to)
8. jl has condition SF ^ OF (less than)
9. jle has condition has condition (SF ^ OF) — ZF (less than or equal to)
10. ja has condition ~CF & ~ZF (unsigned above)
11. jb has condition CF (unsigned below)

12 November 15, 2021

§12.1 Conditional Branches (Continued)

Remark 12.1.1. The %rdi register holds the first argument of a function, the %rsi register holds the second argument, and the %rax register is the return value.

Consult Figure 12.1 to see an example of conditional branch syntax.

```
absdiff:
    cmpq    %rsi, %rdi  # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:        # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Figure 12.1: An example of conditional branching. Notice the jump instruction.

Figure 12.1 shows assembly code not optimized for debugging. If the compiler is left to compile freely, optimizations will be done to express conditionals using **goto code** (like goto statements in C). We write goto code in C during examples because it is easier to see how that code translates directly to assembly.

Example 12.1.2

C code:

```
1 val = Test ? Then_Expr : Else_Expr;
```

Goto version:

```
1 ntest = !Test;
2 if (ntest) goto Else;
3 val = ThenExpr;
4 goto Done;
5 Else:
6     val = Else_Expr;
7 Done: ...
```

Definition 12.1.3. Branches are disruptive to pipelining, so the **conditional move instruction** exists for this purpose. This move supports putting the source inside the destination if something is true. In assembly code, conditional moves are notated with a "c" before the command and the condition afterwards (for example, cmovle).

Remark 12.1.4. In a conditional move, the compiler computes both possibilities (then and else) then returns the correct one to the compiler to avoid branching first.

There are certain cases where conditional moves will not be used by the compiler. These are:

1. Expensive computations
2. Risky computations (computations that may have undesirable effects)
3. Computations with side-effects (e.g. changing variables)

§12.2 Loops

We can perform do-while loops in assembly language by using jump instructions to jump back to the same section of assembly code. Contrast this to conditional branching, where we jump to different sections of the code.

Example 12.2.1 (While Loop General Translation)

C code:

```
while (Test) {
body}
```

Goto version:

```
goto test;
loop:
    Body
test:
    if (Test)
        goto loop;
done:
```

Example 12.2.2 (While Loop General Translation 1)

C code:

```
while (Test) {
Body}
```

Goto version:

```
if (!Test)
    go to done;
loop:
    Body
    if (Test)
        goto loop;
done;
```

To write for loops, we can convert them to while or do-while loops in the typical manner.

13 November 16, 2021

§13.1 Switch Statements

Remark 13.1.1. The same case can have multiple labels. Some cases can end without a break statement, in which case they "fall through" to the next case sequentially. Cases also don't have to be numbered from 1; some numbers can be missing.

Definition 13.1.2. In assembly, a switch statement is achieved via a **jumpe table structure**, which tells the code where to jump to in the case of different values of a register.

Example 13.1.3

```
1 switch_eg:
2     movq    %rdx, ,%rcx
3     cmpq    $6, %rdi # x:6
4     ja      .L8 #default
5     jmp     *.L4(,%rdi,8) #indirect jump
```

The default occurs when the value in the rdi register (the first argument of the function) is above 6. Else, the fourth line makes the code jump to a different address depending on the value in the rdi register. Notice that the base address is at .L4, and each target requires 8 bytes.

Remark 13.1.4. To handle fall-through, we simply jump to the next case inside the current case. This often results in two separate code blocks for fall-through case, one for when the case is entered directly and one when it is entered via another case.

§13.2 Stack Structure

Definition 13.2.1. The **stack** is virtual memory allocated for a function. After the function finishes executing, the stack disappears. The stack grows to lower addresses.

Definition 13.2.2. The **%rsp** register is the stack pointer, and contains the lowest stack address. This is the top element of the stack.

Definition 13.2.3. The **pushq** command pushes the argument onto the stack. The stack pointer is decremented by 8 (since a quad-word is 8 bytes), and the argument is written at the address now given by the stack pointer.

Definition 13.2.4. The **popq** reads the value at the address given by the stack pointer, increments the stack pointer by 8, and stores the value at the given argument. Note that the argument of popq must be a register.

§13.3 Function Control Flow

The stack supports procedure call and return.

Definition 13.3.1. To call a function, we use the **call** command. The **return address**, or the address of the instruction immediately after the call, is pushed onto the stack so the code knows to return to it later, and the code jumps to the label of the function.

Definition 13.3.2. The **ret** command returns from a function. The return address is popped from the stack and the code jumps to it.

§13.4 Passing Data

In order, the first 6 arguments of a function are passed into the rdi, rsi, rdx, rcx, r8, and r9 register. Arguments numbered 7 and onward are passed onto the stack. The return value of the function is placed in the rax register.

14 November 18, 2021

§14.1 Managing Local Data

Definition 14.1.1. An **automatic variable** (or local variable) is a variable declared inside a block or function body. They do not exist outside that block or function, and are created upon entry. Value is not retained across function calls or reinvocations of the block. The value of an automatic variable is stored on the stack.

Definition 14.1.2. A **stack frame** is an instance of the stack for a particular function call.

Stack frames hold return information, local storage (if needed) and temporary space (if needed).

Definition 14.1.3. The %rbp register is the **frame pointer**, pointing to the beginning of the previous stack frame for the current function call or block.

When a function is called, space is allocated by the stack. This includes setting up the code and pushing the return address onto the stack. This memory is deallocated when the function is returned.

> **Remark 14.1.4.** In recursion, each call of the function gets its own stack frame.

§14.1.1 X86-64/Linux Stack Frame

Definition 14.1.5. The **current stack frame** contains an **argument build**, which contains the parameters for the function that about to be called, local variables that are not in registers, saved register context, and he old frame pointer (the rbp register).

Definition 14.1.6. The **caller sack frame** contains the return address and arguments (7 and onwards) for the current calls.

> **Remark 14.1.7.** The area of the stack is bounded by the addresses of the stack pointer and the frame pointer.

Example 14.1.8

Consider the following function in C:

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

In assembly, this is:

```
movq    (%rdi), %rax
addq    %rax, %rsi
movq    %rsi, (%rdi)
ret
```

Consider the following example function:

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

The assembly for this is:

```
subq    $16, %rsp #make space on stack
movq    $15213, 8(%rsp) #move 15213 onto stack
movl    $3000, %esi #second argument in rsi register
leaq    8(%rsp), %rdi #first argument in rdi register
call    incr
addq    8(%rsp), %rax #add together return value to initial value
addq    $16, %rsp #deallocate stack
ret
```

§14.1.2 Register Saving Conventions

Definition 14.1.9. The **caller** is a function that calls another function, which is the **callee**.

We need to make sure contents of registers are not overwritten by callee functions when they are needed in the caller function.

Definition 14.1.10. Some registers are denoted as **caller saved**, so their temporary values are stored in the frame before the call.

Definition 14.1.11. Other registers are denoted as **callee saved**, where the callee saves temporary values in the frame before using, the callee resotres them before returning to the caller.

The caller-saved registers are:

1. %rax
2. %rdi
3. %rsi
4. %rdx

5. %rcx
6. %r8
7. %r9
8. %r10
9. %r11

The callee-saved registers are:

1. %rbx
2. %r12
3. %r13
4. %r14
5. %rbp (Note: This may be used as a frame pointer, but does not have to be).
6. %rsp

Remark 14.1.12. In recursive functions, stack frames mean that each function call gets private storage. Register saving conventions make it so that one function call does not corrupt another. This also works for mutual recursion (P calls Q and Q calls P).

§14.2 Multiple Threads

Definition 14.2.1. A **process** is a running program with its own address space and stack.

Definition 14.2.2. A **thread** is an independently executing function in the same address space as other threads. A new stack is required, but all variables are shared and pointers are valid across threads.

Each thread has its own stack pointer, so it is possible for multiple threads to execute concurrently.

15 November 19, 2021

§15.1 Arrays

Definition 15.1.1. An **array** is a collection of objects of the same type stored contiguously in memory under one name.

> **Remark 15.1.2.** An array of data type T and length L has a contiguously allocated region of size $L \cdot \text{sizeof}(T)$ bytes in the memory.

The identifier of an array (variable name) is used as a pointer to the array element of index 0. Thus, the identifier of an array is of type pointer to the type of object stored.

Example 15.1.3

Take an array cmu of length 5. The rdi register contains the starting address of the array and the rsi register contains the array index. Then, the desired element of the array is at the memory address (%rdi, %rsi, 4).

Example 15.1.4

Here is an example of looping through an array:

```
1 movl    $0, \%eax
2 jmp     .L3
3 .L4:
4     addl    $1, (\%rdi, \%rax, 4)
5     addq    $1, \%rax
6 .L3:
7     cmpq    $4, \%rax
8     jbe     .L4
9     rep; ret
```

§15.1.1 Multidimensional Arrays

A 2D array with R rows and C columns takes up $R \cdot C \cdot \text{sizeof}(T)$ bytes.

Definition 15.1.5. In memory, the array is arranged in **row-major ordering**, so that the element in row r and column c would be contained at index $R \cdot r + c$.

Definition 15.1.6. We can access each **row vector** through starting address $A + i \cdot C \cdot \text{sizeof}(T)$.

Definition 15.1.7. On older systems that did not do multiplication quickly, multidimensional arrays were created via an array of pointers to several one-dimensional arrays. This is a **multi-level array**.

> **Remark 15.1.8.** In a multi-level array, not as much multiplication is required to access elements, but two memory reads must be done, first to the pointer, then to the element within the corresponding array.

If dimensions are known at compile time, then the assembler can retrieve array elements quickly and explicitly with shifts instead of multiplication. If dimensions are variable but are passed into a function, multiplication must be used, which is slightly slower.

§15.2 Structs

A struct is represented as a block of memory big enough to hold all of the fields. Fields are ordered according to declaration, even if another ordering would be more compact. The compiler determines the size and positions of these fields (the machine-level program does not have knowledge of this).

16 November 22, 2021

§16.1 Structs (Continued)

Example 16.1.1

Say we want to copy a linked list into an array. We have a struct with an int array (say, of 4 elements), the current element of the linked list, and a pointer to the next element. Then, the C code and corresponding assembly is:

```
1  struct rec {
2  int a[4];
3  int i;
4  struct rec *next;
5  };
6
7  void set_val(struct rec *r, int val) {
8  while (r) {
9  int i = r->i;
10 r->a[i] = val;
11 r = r->next;
12 }
13 }
```

```
1  .L11:
2      movslq   16(\%rdi), \%rax #i = M[r+16]
3      movl     \%esi, (\%rdi, \%rax, 4) #M[r+4*i] = val
4      movq     24(\%rdi), \%rdi #r = M[r+24]
5      testq    \%rdi, \%rdi #Test r
6      jne      .L11 #if !=0 goto loop
```

Definition 16.1.2. In a struct, data can be unaligned since data types require different amounts of space. Then, the compiler will **align** the data by inputting blank space so that each address is a multiple of K, where K is the size of a primitive data type in bytes.

Definition 16.1.3. The **alignment requirement** of a datatype is the number of bytes it takes up in memory.

Example 16.1.4

If the largest primitive data type is char, there are no restrictions on alignment. If the largest is a short, then the lowest 1 bit of the address must be a 0. This pattern continues for 4 byte data types (int, float), 8 byte data types (double, long, pointers), and 16 byte data types (long double).

Within the structure, the structure must satisfy each element's alignment requirement. Overall, the structure has alignment requirement K, where K is the largest alignment of any element. Thus, the initial address and structure length must be multiples of K.

If we have an array of structs, then the overall structure length is a multiple of K, where K is the largest alignment requirement. Each element of the array also satisfies

the alignment requirement. Structs within the array are accessed as usual, except with array offset equal to the size of the struct, including alignment spacers.

> **Remark 16.1.5.** To save space in a struct, put large data types first so there is less padding.

Definition 16.1.6. A **union** is a struct, except the same starting address is used for each member of the union. Thus, you can only use one field at a time in a union.

> **Remark 16.1.7.** Unions are essentially only when you are trying to be very space-efficient.

§16.2 More on Floating Point

Definition 16.2.1. Floating point operations are carried out in the **XMM registers**. There are 16 of these registers total, each 16 bytes.

Arguments are passed into %xmm0, %xmm1, and so on. The result is returned in %xmm0. All of the xmm registers are caller-saved.

> **Remark 16.2.2.** Integers and pointers are still passed in regular registers. There are different move instructions to move between xmm registers and between memory and xmm registers (movapd, movsd respectively).

Definition 16.2.3. Floating-point comparisons are done via the **ucomiss, ucomisd** commands. These set the condition codes CF, ZF, and PF.

> **Remark 16.2.4.** The XMM0 register can be set to 0 using the instruction xorpd %xmm0, %xmm0.

§16.3 Memory Layout

The x86-64 Linux memory layout is a contiguous vector. There is a stack that begins from the top and grows to lower addresses, as discussed earlier. There are also other portions of memory:

Definition 16.3.1. The **heap** starts in lower memory, and is where dynamically allocated memory is stored (malloc, calloc, new).

Definition 16.3.2. The **data** area holds statically allocated data like global variable, static variables, and string constants.

Definition 16.3.3. The **text** area holds text and shared libraries (commonly used programs and commands). An example of a shared library is the printf command.

17 November 23, 2021

§17.1 Buffer Overflow

Example 17.1.1

Consider the following struct and function:

```
typedef struct {
int a[2];
double d;
} struct_t;
```

```
double fun(int i) {
volatile struct_t s;
s.d = 3.14;
s.a[i] = 1073741824;
return s.d;
}
```

We get fun(0) = 3.14, fun(1) = 3.14, fun(2) = 3.1399998664856, fun(3) = 2.00000061035156, fun(4) = 3.14, and fun(6) throws a segmentation fault.

To understand this, note that the double is allocated in the next two 4-byte spaces beyond the two elements of the int. Thus, fun(2) and fun(3) allocate the array to the space that contains the double, changing its value. Fun(4) is outside the scope of the struct, so the return value is fine. However, fun(6) changes the return address or some other vital piece of information, causing the program to blow up.

Definition 17.1.2. The above problem (exceeding the memory size allocated for an array) is called a **buffer overflow**.

The most common form of this vulnerability is unchecked length of a string input.

Remark 17.1.3. The Unix function gets() suffers from potential buffer overflow, as there is no way to specify the limit of the number of characters to read.

Example 17.1.4

Consider the following C code and associated assembly code, which prints a 24-character string fine but blows up at 25 characters:

```
void echo() {
char buf[4];
gets(buf);
puts(buf);
}

void call_echo() {
echo();
}
```

```
echo:
sub     $0x18,\%rsp
mov     \%rsp,\%rdi
callq   gets
mov     \%rsp,\%rdi
callq   puts
add     $0x18,\%rsp
retq

call_echo:
sub     $0x8,\%rsp
mov     $0x0,\%eax
callq   echo
add     $0x8,\%rsp
retq
```

Echo moves the stack down 24 bytes, so there is enough space for 24 characters in the string before the resulting buffer overflow results in a fatal error by overlaying part of the return address (this causes a segmentation fault).

Definition 17.1.5. If an input string contains a byte representation of executable code, then the return address can be overwritten with a buffer, which will cause the return statement to access unwanted memory. This is a **code injection attack**.

Remark 17.1.6. An example of a code injection attack based on buffer overflow is the original "internet worm" in 1988. The worm exploited the gets function and invaded about 6000 computers in hours.

§17.1.1 Protection Against Buffer Overflow Attacks

The first, most obvious way to do this is to avoid vulnerabilities by using functions that limit string lengths, like fgets or strncpy. Additionally, do not use scanf with a %s conversion specification.

Another method is system-level protection:

Definition 17.1.7. At the start of the program, a random amount of space is allocated on the stack. This shifts the stack addresses and makes it difficult for the hacker to predict the beginning of the inserted code. This is **stack randomization**.

Definition 17.1.8. Certain code segments are labeled as **nonexecutable**, or read-only, to protect against execution of certain parts of code.

The final method is a stack canary:

Definition 17.1.9. A special value (**stack canary** is placd on the stack just beyond the buffer. This checks for corruption before exiting the function, since if it is changed, we know corruption has occurred. This canary is in the fs register.

18 November 29, 2021

§18.1 Return-Oriented Programming Attacks

This type of attack uses existing code and strings together fragments to achieve the desired outcome. Importantly, this does not overcome stack canaries.

Definition 18.1.1. The fragmented program is constructed from **gadgets**, which are sequences of instructions ending in ret.

Gadgets can be constructed from tail ends of existing functions or by repurposing byte codes.

§18.2 The Memory Hierarchy

So far, we have been dealing with the virtual memory of the program. Now, let's take a look at how memory is really stored. Figure 18.1 shows the design of a modern processor.

Definition 18.2.1. A **superscalar processor** can issue and execute multiple machine instructions in one cycle (roughly a third of a nanosecond). They are received from a sequential instruction stream.

> **Remark 18.2.2.** Superscalar processors can take advantage of inherent instruction-level parallelism present in most programs.

> **Remark 18.2.3.** Most modern processors are superscalar processors.

Instructions can be **pipelined** with superscalar processors. For example, with three parallel processing units, 3 multiplications can be completed in much less than 9 cycles, even though each takes 3 cycles on its own.

Consider adding. If we add from register to register, one cycle is needed as registers can be fetched almost immediately. However, if we add to memory addresses, we need many cycles since we are retrieving from memory.

Definition 18.2.4. **On-chip** memory, or static RAM, like registers and caches, takes 2-10 cycles to retrieve.

Definition 18.2.5. **Off-chip** memory, or dynamic RAM (values at generic addresses) takes 100+ cycles to retrieve.

Thus, we need quick access to memory to fully utilize the added computing power of superscalar processors. This means on-chip memory should be used as much as possible.

§18.3 Locality

Definition 18.3.1. The **principle of locality** states that programs tend to use data and instructions near those they have used recently.

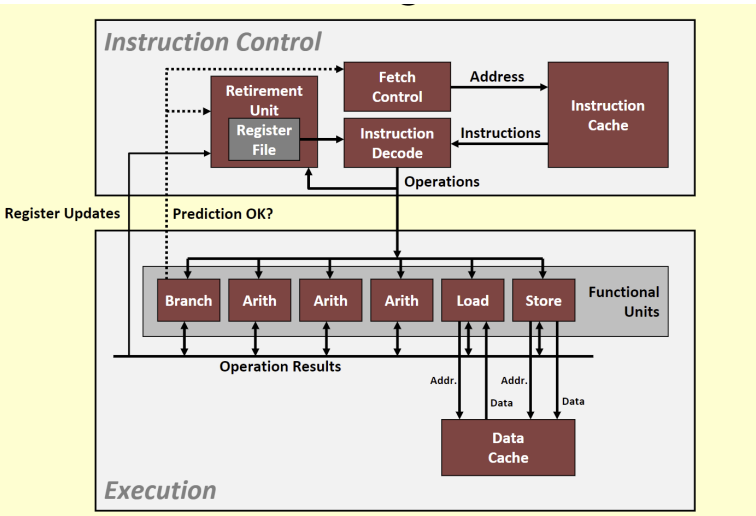

Figure 18.1: The design of a modern processor.

Definition 18.3.2. **Temporal locality** states that recently referenced items are likely to be referenced again soon.

Definition 18.3.3. **Spatial locality** states that items nearly recently referenced items are likely to be referenced soon.

Example 18.3.4

Consider looping through the elements of a 2D array. Since elements are ordered in row-major order, good locality would be to loop through the columns of each row (spatial locality). However, looping through the rows of each column would be much slower and have very poor locality.

19 November 30, 2021

§19.1 Caches

Definition 19.1.1. A **cache** is a small, fast memory that holds a subset of items from a larger, slower memory. This approximates the performance of the fast memory while retaining the size of the larger memory.

When designing a program, one should design a cache system so that most references occur on the cache and are thus fast.

Example 19.1.2

In transaction processing, records of today's departures are on caches, while records of future flights are on remote databases.

Definition 19.1.3. **Virtual memory** is the idea that each running program has its own memory, separate from all other running programs, computers, or users. These memories are actually stored on the disk.

Definition 19.1.4. **RAM** is a cache of the frequently used parts of virtual memory.

Definition 19.1.5. A **Cache hit** is when a memory access resolves to an item in the cache, resulting in fast access.

Definition 19.1.6. A **Cache miss** is when a memory access does not resolve to anything in the cache.

Caches throw out not commonly used items in the event of misses. This is often by which item was least recently used (LRU).

Caches can be layered as well.

Example 19.1.7

1. The L1 cache is a cache of L2 memory that is 32 kilobytes large and takes 4 cycles to access.
2. L2 is a cache of L3 memory that is 256 kilobytes large and takes 11 cycles to access.
3. L3 is a cache of DRAM which is 8 megabytes large and takes 30-40 cycles to access.
4. DRAM is a cache of virtual memory and takes hundreds of cycles to access.

Definition 19.1.8. Thus, the memory hierarchy is as follows ,where higher levels take longer to access and also hold more memory:

1. Level 0 consists of registers.
2. Level 1 consists of the L1 cache (SRAM).

3. Level 2 consists of the L2 cache (SRAM).
4. Level 3 consists of the L3 cache (DRAM).
5. Level 4 consists of main memory (DRAM).
6. Level 5 consists of local secondary storage (local disks).
7. Level 6 consists of remote secondary storage like web servers.

Note that each level is a cache of the level above.

§19.1.1 Cache Performance

Definition 19.1.9. The **miss rate** is the fraction of memory references not found in the cache. This tends to be 3 to 10 percent for the L1 cache and less than one percent for L2 and beyond.

Definition 19.1.10. The **hit time** is the time it takes to deliver a block in the cache to the processor. For L1, this tends to be 1 to 4 cycles, and in L2 it tends to be 10 to 20 cycles.

Definition 19.1.11. The **miss penalty** is the additional time required because of a miss. This time is used to search and retrieve from lower level memory, and tends to be around 50 to 200 cycles.

Theorem 19.1.12

The average access time for memory from a certain cache is the hit time plus the miss rate times the miss penalty.

To calculate the averge access time of memory overall, we can perform this calculation down the memory hierarchy.

Remark 19.1.13. These same principles, in this section and the previous, can be applied to multi-level memory.

Definition 19.1.14. A **cold miss** is when the cache is empty.

Definition 19.1.15. Most caches make it so that blocks in a cache can only map to a certain subset of blocks in the cache a level above. A **conflict miss** occurs when the cache is large enough, but multiple data objects all map to the same block in the cache.

Definition 19.1.16. A **capacity miss** occurs when the working set of blocks is larger than the size of the current cache.

20 December 3, 2021

§20.1 Cache Memories

Definition 20.1.1. **Cache memories** are small and fast SRAM memories managed automatically in hardware. They hold frequently accessed blocks of memory.

> **Remark 20.1.2.** Processors look first for data in caches, then in main memory if it can't find it.

Definition 20.1.3. Caches are organized into **sets, lines, and bytes**. Specifically, a cache has $S = 2^s$ sets, $E = 2^e$ lines per set, and $B = 2^b$ bytes per line for some integers s, e, b. Thus, the size of a cache is $S \cdot E \cdot B$ bytes.

Definition 20.1.4. Each line of a cache contains B bytes of data, a **valid bit** v that checks that the associated memory in the line is not from a previous program, and a set of t identifying bits known as the **tag**.

The address of a word in a cache consists of t bits for the tag followed by s bits for the set index and b bits for the block offset. To check if a word is in a cache, the machine first checks the set index to see if the set is in the cache. Then, it checks if any line in the set has a matching tag. If there is a matching tag and the corresponding line is valid, we have a cache hit. Then, data is located using the last b bits of the address, which is the block offset in the line.

Definition 20.1.5. A **direct mapped cache** is when $E = 1$, so there is one line per set.

> **Remark 20.1.6.** Data is read into a cache by lines.

Definition 20.1.7. A **cache eviction** is when data is removed from the cache, typically after a conflict miss.

Example 20.1.8

Say we begin with a direct mapped cache with $t = 1, s = 2, b = 1$. Then, reading in the following addresses will cause:

1. 0 will result in a miss since the cache is empty. Set 0 will read in 0 and 1.
2. 1 will result in a hit.
3. 7 will result in a miss. Set 3 will read in 6 and 7.
4. 8 will result in a miss since there is a conflict miss with 0 and 1. The cache will read in 8 and 9 and evict 0 and 1 from set 0.
5. 0 will result in a conflict miss with 8 and 9. The cache will evict 8 and 9 and read in 0 and 1.

Notice how cache misses end up filling the cache block.

Definition 20.1.9. An **E-way Set Associative Cache** is a cache with E lines per set.

Remark 20.1.10. If a cache has more than one line per set, then tag bits come into play when determining conflict misses. If eviction is necessary, one line must be selected for eviction. This can be done via random policy, least recently used policy, or others.

Remark 20.1.11. In the direct mapped cache example, if the cache was a 2-way set associative cache, the last 0 would result in a cache hit rather than a miss.

21 December 6, 2021

§21.1 Finishing Caches

Note that multiple copies of data exist, down the memory hierarchy.

Definition 21.1.1. A **write-through** writes data to a cache then copies it down the memory hierarchy.

Definition 21.1.2. A **write-back** defers writing to memory until a line is replaced. This needs a dirty bit to confirm whether a line is different from the rest of memory.

Definition 21.1.3. A **write-allocate** is when a line is loaded into a cache then updated.

Definition 21.1.4. No write-allocate means the line is written immediately to memory.

> **Remark 21.1.5.** Typically, processors use write-back and write-allocate for write hits and write misses, respectively.

§21.2 Cache Performance

§21.2.1 Memory Mountain

Definition 21.2.1. **Read throughput** is the number of bytes read from memory per second.

Definition 21.2.2. The **memory mountain** measures read throughput as a function of spatial and temporal locality.

Both spatial and temporal locality have a large effect on performance.

§21.2.2 Rearranging Loops To Improve Locality

Consider a cache with line size 32 bytes, and say we want to multiply two large N by N matrices, so large that the cache is not big enough to hold multiple rows. To improve performance, we can consider the access pattern of the inner loop.

Example 21.2.3

```
  1 double sum;
2 int i,j,k;
3 for (i=0; i<n; i++) {
4     for (j=0; j<n; j++) {
5         sum = 0.0
6         for (k=0; k<n; k++) {
7             sum += a[i][k] * b[k][j];
8         }
9         c[i][j] = sum;
10    }
11 }
```

On average, we have a miss rate of 0.25 per inner loop iteration of matrix A, 1.0 for B, and 0.0 for C.

If we instead use a jik loop order, we get the same miss rates. However, a jki/kji order is worse, giving 1.0 miss rate for A and C and 0.0 for B. A kij/ikj order is the best, giving 0.25 for B and C and 0.0 for A.

§21.2.3 Blocking

Definition 21.2.4. Another way to reduce cache misses is **blocking**. This is when we reduce a function into blocks of data that fit into the cache, and only consider blocks at a time.

22 December 7, 2021

§22.1 Linking

Example 22.1.1

Consider two functions.

```
int buf[2] = {1,2};

int main() {
swap();
return 0;}
```

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap() {
int temp;

bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

Definition 22.1.2. **Static linking** is when programs are translated and linked using a **compiler driver**, such as gcc.

Example 22.1.3

```
linux> gcc -O2 -g -o p main.c swap.c
linux> ./p
```

First, the source files main.c and swap.c are run through a translator to generate separately compiled object files main.o and swap.o. Then, the **linker** puts the files main.o and swap.o together to create a linked executable object file.

> **Remark 22.1.4.** We use linkers so that programs can be written as a collection of smaller source files, rather than one big one. It is also more efficient, since changing one source file does not require everything else to be changed.

Definition 22.1.5. Linkers perform **symbol resolution**, which is connecting declared objects with references to those objects in other modules.

Definition 22.1.6. Linkers perform **relocation**, which is repositioning code within the executable image and changing the values of pointers accordingly.

§22.1.1 Symbol Resolution

Programs define and references symbols, which are variables and functions.

Example 22.1.7

```
1 void swap() { } /* define symbol "swap" */
2 swap(); /* reference symbol "swap" */
3 int *xp = &x; /* define symbol "xp", reference symbol "x" */
```

Definition 22.1.8. Symbols are stored by the compiler in a **symbol table**. This is an array of structs where each entry has the name, size, and location of what the symbol refers to.

During symbol resolution, the linker associates each symbol reference with exactly one symbol definition.

§22.1.2 Relocation

Linkers merge code, data sections and all sections of the same type into new sections of the same type during relocation. Then, symbols are relocated from their locations in the .o files to their final positions in the executable, updating references as necessary.

Object Files

There are three types of object files:

Definition 22.1.9. A **relocatable object file** is a .o file. This contains code and data that can be relocated to form an executable.

Definition 22.1.10. An **executable object file** is a .out file. This contains code in a form that can be immediately copied and executed.

Definition 22.1.11. A **shared object file** is a .so file. This is a relocatable object file that is loaded and lnked dynamically (at either load or run time). Windows calls this a **Dynamic Link Library**.

§22.1.3 ELF Format

These object files have a standardized format called **Executable and Linkable Format (ELF)**. The name for these files is ELF binaries.

Definition 22.1.12. ELF format is as follows, in order from top to bottom:

1. The **ELF header** contains the word size, byte ordering, file type, machine type, etc.
2. The **segment header table** contains the page size, virtual addresses to memory segments, segment sizes.
3. The **.text section** contains code.
4. The **.rodata section** contains read only data such as jump tables, vtables, format strings, and so on.
5. The **.data section** contains initialized global and static variables.
6. The **.bss section** contains uninitialized global and static variables.

7. The **.symtab section** contains the symbol table, procedure and static variable names, and section names and locations.
8. The **.rel.text section** contains relocation info for the .text section as well as instructions for performing the relocating.
9. The **.rel.data section** contains relocation info for the .data section.
10. The **.debug section** contains info for symbol debugging (i.e. gcc -g).
11. The **section header table** contains the offsets and sizes of each section.

§22.2 Linker Symbols

Definition 22.2.1. **Global symbols** are symbols defined by a module m that can be referenced by other modules, like non-static global variables.

Definition 22.2.2. **External symbols** are global symbols that are referenced, but not defined, in module m.

Definition 22.2.3. **Local symbols** are symbols that are defined and referenced exclusively by module m.

> **Remark 22.2.4.** Local symbols are NOT local progam variables!

23 December 9, 2021

§23.1 Symbols (Cont.)

Definition 23.1.1. **Strong symbols** are procedures/functions and initialized globals.

Definition 23.1.2. **Weak symbols** are uninitialized global variables.

The linker has several rules when dealing with symbols:

1. Multiple strong symbols of the same name are not allowed.
2. If there is one strong symbol and multiple weak symbols of the same name, references to the weak symbol will resolve to the strong symbol.
3. If there are multiple weak symbols with the same name, we pick an arbitrary one to resole all references to. This can be overridden with the gcc -fno-common command.

§23.2 Header Files

Definition 23.2.1. A **header file** is a text file containing C code that defines/identifies functions, variables, macro definitions, etc.

The results of linking can depend on whether or not certain variables in the .h file are used (can initialize/not initialize certain global variables, resulting in different combinations of strong and weak symbols).

§23.3 Static Libraries

Question 23.3.1. How do we package functions commonly used by programmers? (math, I/O, etc.)

We can eithe rput all the functions into one source file (space and time inefficient) or put each function in a separate source file (more efficient, but big burden on the programmer to link each binary separately).

The solution is a static library!

Definition 23.3.2. A **static library** is a .a archive file. This concatenates related relocatable object files into a single file with an index (this is called an **archive**). Then, the linker tries to resolve unresolved external references by looking for the symbols in archives. If found, the linker links the appropriate symbol into the executable.

> **Remark 23.3.3.** Commonly used libraries are libc.a (standard C library) and libm.a (C math library).

24 December 10, 2021

§24.1 Static Libraries (cont.)

Definition 24.1.1. An **archiver**, or .ar file, is a file that puts together several executable object files for each function to be in the archive. This is what creates a static library and allows for incremental updates.

To link static libraries, the linker:

1. Scans .o and .a files in command line order.
2. During scan, keep list of current current unresolved references.
3. When a new .o or .a file is encountered, each unresolved reference is attempted against the symbols defined in the file.
4. If there are any unresolved references at the end, error.

> **Remark 24.1.2.** This means command line order matters when linking! Put libraries at the end.

§24.2 Loading Executable Object Files

Consult Figure 24.1 to see how an ELF file is loaded into virtual memory:

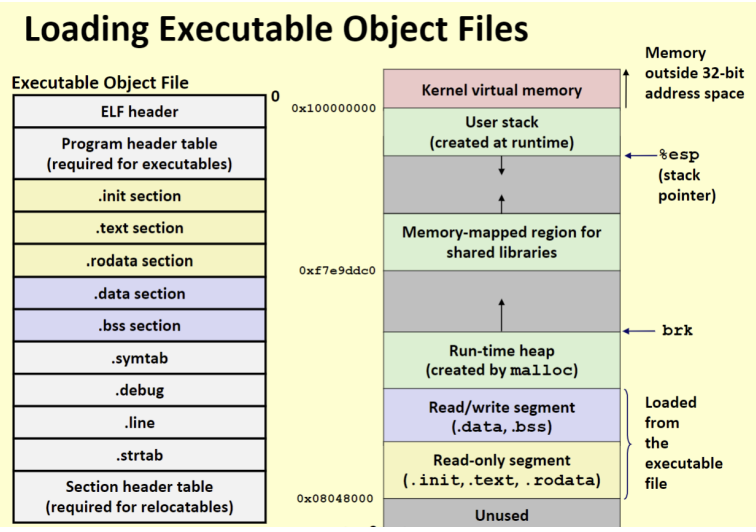

Figure 24.1: How an ELF file is loaded in.

§24.3 Shared Libraries

Static libraries have some disadvantages:

1. They take up a lot of space on the disk.
2. Each application must explicitly relink for every minor bug fix of the library.

Definition 24.3.1. A **shared library** is an object file that is loaded and linked dynamically. These are also called **dynamic link libraries**, or DLLs, and are .so files.

This dynamic linking occurs when an executable is first loaded and run (**load-time linking**). It can also occur after the program has begun (**run-time linking**).

> **Remark 24.3.2.** Shared library routines can be shared by multiple processes.

Definition 24.3.3. To link at load-time, the linker first checks that the references to the .so file can be resolved. Then, the partially-linked executable object file (without the .so file) is put into a **loader** to resolve references to functions in the .so file.

§24.4 Virtual Memory

§24.4.1 Address Spaces

Definition 24.4.1. **Physical addressing** is when addresses explicitly point to a part of memory that is physical (high-voltage/low-voltage switch, etc.)

Definition 24.4.2. **Virtual addressing** is when addresses point to an address in **Memory Management Unit** (MMU), which points to a place in physical memory. A program believes it owns virtual memory.

Definition 24.4.3. **Linear address space** is an ordered set of contiguous nonnegative integer addresses ($\{0, 1, \ldots\}$)

Definition 24.4.4. **Virtual (logical) address space** is a set of $N = 2^n$ virtual addresses. This is what the program sees.

Definition 24.4.5. **Physical address space** is a set of $M = 2^m$ virtual addresses. This is what the hardware uses.

What this means is that every byte in main memory has one physical address and one ore more virtual addresses. In other words, every object can now have multiple addresses.

Virtual Memory allows for main memory to be used efficiently, since it only loads parts in usage. Memory management is simpler, since each process gets the same uniform linear address space. Finally, it isolates address spaces, so one process doesn't interfere with another.

25 December 13, 2021

§25.1 Uses of VM

§25.1.1 Caching

Virtual memory is an array of contiguous bytes on the disk. The contents of this array are cached in physical memory, or DRAM. Of course, this memory is then cached into higher levels of the memory hierarchy, which is all in physical memory.

Definition 25.1.1. The cache blocks of physical memory are called **pages**.

> **Remark 25.1.2.** DRAM is about 10x slower than SRAM, but the disk is about 10000x slower than DRAM! This leads to large page sizes of 4 KB to 4 MB. The cache is also fully associative, so any virtual page can be placed in any physical page.

> **Remark 25.1.3.** The cache is write-back.

Definition 25.1.4. The **page table** is an array of page table entries (PTEs) which map virtual pages to physical pages.

Definition 25.1.5. A **page hit** is a reference in the page table that is in physical memory.

Definition 25.1.6. A **page fault** is a reference in the page table that is not in physical memory.

A page fault causes a random page to be evicted, the new page is allocated in virtual memory, and the offending instruction is restarted. This is very expensive!

> **Remark 25.1.7.** Locality makes virtual memory work, since programs tend to reference similar memory locations.

Definition 25.1.8. The **working set** is the set of active virtual pages.

Definition 25.1.9. **Thrashing** can occur when the sum of all working set sizes is greater than the main memory size. This causes a meltdown where pages are continuously copied in and out.

§25.1.2 Memory Management

The KEY IDEA is each process has its own virtual address space! A mapping function scatters addresses through physical memory so they don't overlap, but each process views memory as a simple linear array.

To share code among processes, virtual pages can even be shared to the same physical page!

Virtual memory also simplifies linking and loading. The code, stack, and shared libraries always start at the same address, simplifying linking.

When loading executable files, the .text and .data sections of the ELF file are copied on demand by VM into the main memory (allocated then copied).

§25.1.3 Memory Protection

Page table entries can be extended with permission bits that are checked before remapping. This allows certain sections of memory to be protected.

§25.2 Address Translation

Recall that virtual address space is given by the set $V = \{0, 1, \ldots, N-1\}$, where there are $N = 2^n$ addresses in virtual address space. Similarly, physical address space is given by $P = \{0, 1, \ldots, M-1\}$ where there are $M = 2^m$ addresses in physical address space.

Definition 25.2.1. Addresses are translated via a **map function** MAP from V to $P \cup \{\emptyset\}$. For a given $a \in V$, we say $MAP(a) = a'$ if the data at virtual address a is at physical address a'. If the data at virtual address a is not in physical memory (invalid/on disk), then $MAP(a) = \emptyset$.

Definition 25.2.2. The following are the main components of the virtual address:

1. **TLBI**: Translation Lookaside Buffer Index
2. **TLBT**: Translation Lookaside Buffer Tag
3. **VPO**: Virtual Page Offset
4. **VPN**: Virtual Page Number

Definition 25.2.3. The following are the main components of the physical address:

1. **PPO**: Physical Page Offset
2. **PPN**: Physical Page Number
3. **CO**: Byte offset in cache line
4. **CI**: Cache Index
5. **CT**: Cache Tag

To translate addresses with a page table, the virtual address is inputted. The high order $n-p$ bits comprise the VPN and the low order p bits comprise the VPO. Through the page table, the VPN is converted into an $m-p$-bit PPN, and the VPO is converted directly into a PPO without changing its value. This creates an m-bit physical address.

In the case of a page hit, the processor:

1. Sends the virtual address to the MMU
2. The MMU fetches the PTE from the page table

3. The MMU sends the physical address to the cache
4. The cache sends the data to the processor

In the case of a page fault, the processor:

1. Sends the virtual address to the MMU
2. The MMU fetches the PTE from the page table
3. The valid bit is zero, so the MMU triggers an exception
4. The handler identifies a page to evict and pages in a new page, updating the PTE as appropriate
5. Original process restarts

§25.2.1 Translation Lookaside Buffer

PTEs are usually cached in L1, so they could be evicted and have a small delay. The solution is a TLB.

Definition 25.2.4. A **Translation Lookaside Buffer (TLB)** is a small cache in the MMU that maps VPNs to PPNs. It contains complete PTEs for a few pages.

To access the TLB, the VPN is split into a TLBT and a TLBI. The TLBI functions as a set index and the TLBT functions as a tag check.

In the case of a TLB hit, the TLB gets rid of a memory access, speeding up the process. However, if the TLB misses, then the PTE must be accessed, taking longer.

> **Remark 25.2.5.** Page tables are often multiple levels due to their very large size.